CyPro User Manual

rev. 201

Robotina d.o.o.
Industrijska cesta 2b
6310 Izola, Slovenia
tel: ++386 5 663 2420
fax: ++386 5 663 2439
e-mail: vcp@robotina.si
http://www.cybro-plc.com
AP-CYP-700-201-180302

Index

Index 2
Introduction 5
System overview 5
Hardware requirements 6
Installation 6
Communication 7
User interface 8
Main window 8
Standard toolbar 8
Communication toolbar 9
Project tree 9
Status bar 10
Menu 10
File 10
Edit 11
View 11
Insert 11
Communication 11
Options 11
Edit window 12
Edit instruction list 12
Edit structured text 13
Variable monitor 14
Snapshot 14
Programming 16
Hardware 16
IEX expansion modules 16
Hardware setup 17
Variables 18
Naming convention 18
Allocation 18
Initialization 19
I/O variables 19
Internal variables 21
Timers 22
Pulse timer 22
On-delay timer 23
Counters 23
Program execution 24
Instruction list 25
Load and store instructions 27
Logical instructions 27
Edge instructions 30
Arithmetic instructions 31
Comparison instructions 32
Type conversion instructions 32
Branching instructions 34
Structured text 35
Assignment statements 35
Expressions 35
Operators 35
Expression evaluation 36
Data type conversions 36
Multiline expressions 36
Conditional statements 37
if...then...else 37
case...of 38

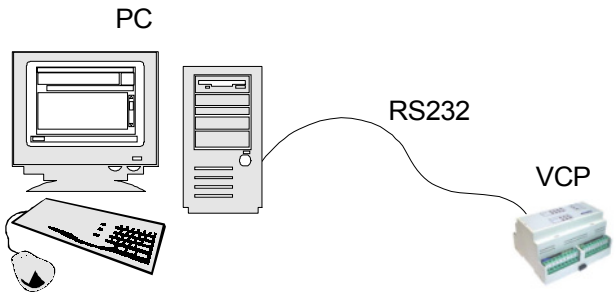
Iteration statements ... 38
for...do ... 38
while...do ... 39
Operator panel ... 40
General ... 40
Print functions ... 40
OP keys ... 42
Masks ... 43
Com2 port ... 46
General ... 46
Initialization ... 46
Prepare transmit message ... 47
Transmit ... 48
Receive ... 48
Parse received message ... 50
High speed counter ... 52
General ... 52
High speed counting ... 52
High speed action ... 53
Zero reset ... 55
Zero detect ... 56
Real-time clock ... 57
General ... 57
RTC read ... 57
RTC adjust ... 58
Networking ... 59
General ... 59
Sockets ... 60
Debug mode ... 62
Network example ... 63
Network performance ... 65
VCP tutorial ... 69
Your first VCP program ... 69
Step one: define problem ... 69
Step two: select hardware ... 69
Step three: allocate variables ... 70
Step four: write code ... 70
Step five: bring it to life ... 70
Step six: new frontiers ... 71
Appendix ... 73
Data types summary ... 73
Elementary ... 73
Input/Output ... 73
Timer ... 73
Counter ... 73
Internal variables ... 74
Instruction list summary ... 75
Instructions ... 75
Allowed type conversions ... 76
Allowed type combinations ... 76
Structured text summary ... 77
Operators ... 77
Statements ... 77
Edge detect functions ... 78
Cast functions ... 78
Display functions ... 78
Com2 functions ... 79
High speed counter functions ... 79
Real-time clock functions ... 80
Error messages ... 81

PLC status 81
Compiler messages 81
Allocation messages 82
Mask messages 82
Socket messages 82
Keyboard shortcuts 83
Common 83
Text editor 83

Introduction

System overview

CyPro is a software package for programming VCP controllers. It runs under Microsoft Windows 95/98/ME/NT4/2000. CyPro is a fully featured IDE (integrated development environment) containing an editor, a compiler and an on-line monitor.



VCP is the generic name for a group of controllers, based on VCP technology, with a variety of final products. All VCP models are software compatible, although they may differ in other features. Each VCP is labeled with the unique 6-digit serial number, also used in communication as the network address.

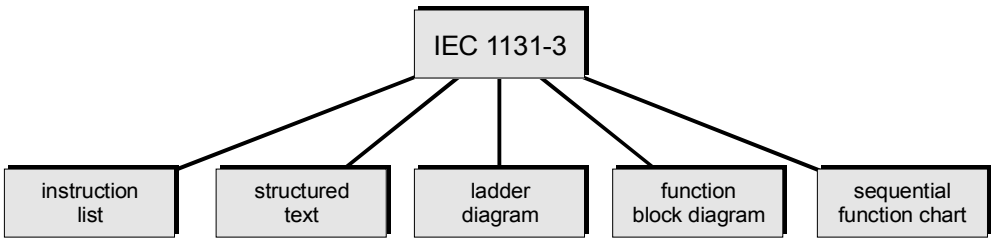
Pico

CyBro

A

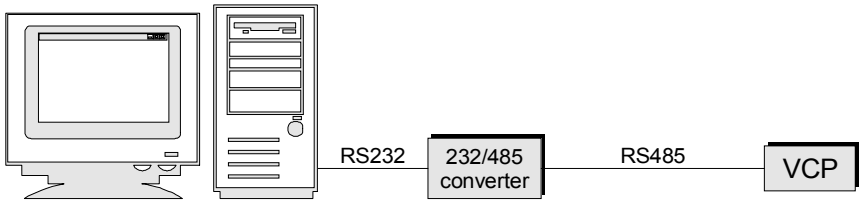
Every VCP has at least two communication channels, one for programming and networking, and another for connecting expansion boards. The number of the expansion slots depends on the VCP model used.

CyPro is based on IEC 1131-3 standard. Current version implements instruction list and structured text programming, extended with several useful visual tools.



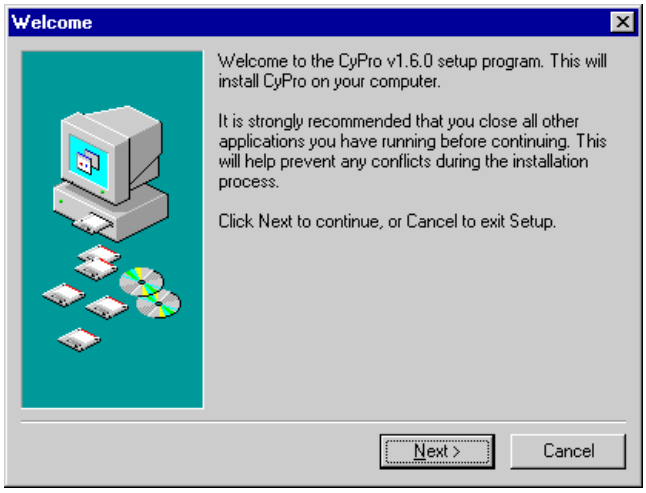
Hardware requirements

Any PC capable of running MS Windows 95 is satisfactory, although the recommended minimum is Pentium 166 with 32Mb RAM. CyPro will occupy about 2Mb of the disc space. A standard RS232 COM port is required for program transfer and on-line monitor. Some VCPs are equipped with the RS485 serial port only, and in such case, the external RS232/RS485 converter is required.



Installation

To install CyPro, simply start the installation archive and follow the instructions. The recommended install directory of CyPro is C:\Program Files\CyPro.


Welcome
Welcome to the CyPro v1.6.0 setup program. This will install CyPro on your computer.
It is strongly recommended that you close all other applications you have running before continuing. This will help prevent any conflicts during the installation process.
Click Next to continue, or Cancel to exit Setup.
Next >
Cancel

Installation does the following:

- unpacks CyPro files into specified directory
- creates start menu group and inserts icons
- sets association to .cyp file type
- sets CyPro language and communication port

No files are copied in the windows directory. No windows system files will be replaced or changed. Default directory for user projects is C:\Program Files\CyPro\Project, but the user can Open or Save projects in any other directory.

To update CyPro, install a new version into the same directory. CyPro system files will be updated and the user settings will be preserved.

With the new CyPro release, the VCP system (kernel) should also be updated. To do this, start Communication/Kernel Maintenance, load kernel.bin and send it to the VCP. A new kernel is located in the same directory where the CyPro is installed.

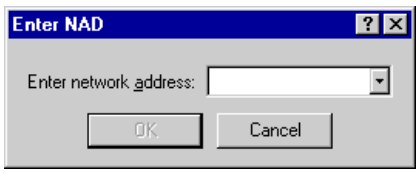
To uninstall CyPro, start Control Panel, Add/Remove Programs, select CyPro and press Add/Remove button.

Communication

The serial RS232 communication is used for transferring compiled program and for monitoring VCP on line. The communication indicators will be displayed on the status bar.

In order to communicate with the VCP, CyPro has to know its network address. Network address is a unique 6-digit number labeled on each VCP.

To enter network address start any communication command. Enter NAD dialog box will appear. Type the address of your VCP and press **Enter**. The other posibility is to right click the network address indicator on the status bar, or to select Options/Project/Network address and insert the network address of the target VCPs.



To select communication port open Options/Environment/Communication and choose port. Other communication parameters (baud rate, number of data and stop bits) are fixed and should not be changed.

In case that communication is not working, check cable indicator icon on the status bar first.

If the cable fails, it is possible that the com port number, the communication cable or connections is invalid. Select the right communication port and check cable and connections.

If the cable is properly detected, check network address, communication cable, connections, or replace VCP.

Note that the status bar is updated around every 3 seconds, immediately after the transmit led flashes.

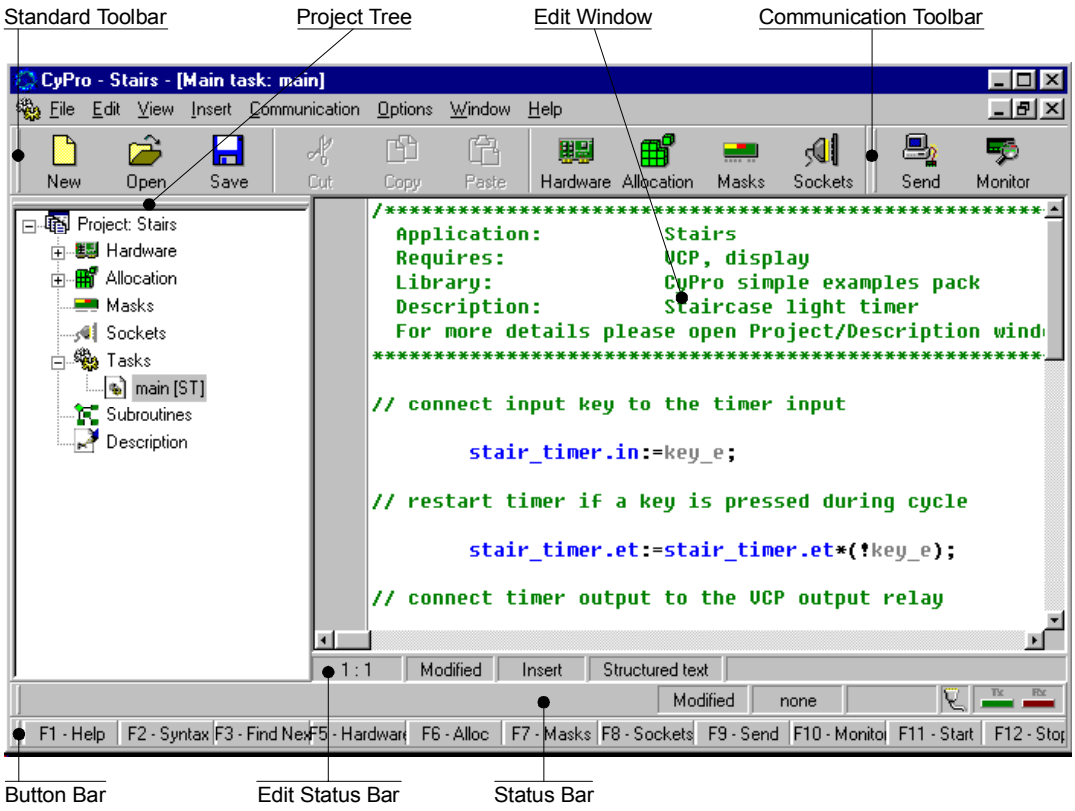
For wiring diagram please check the hardware manual.

If option Synchronize program with VCP is activated, commands Start and Monitor will automatically compile and send the current project to the VCP. The option is located in Options/Environment/General.

User interface

Main window

Main CyPro window consists of edit window, toolbars and status bars. Default positions are shown below:



Every component can be docked or floating. To undock, drag the component by the left vertical line (top horizontal for project tree) over the edit area. To dock again, drag window close to the main window border.

Standard toolbar

Create a new empty project.

Open an existing project.

Save the current project.

Remove the selection and place it on the clipboard.

Copy the selection onto the clipboard.

Insert the clipboard at the cursor, replacing any selection.

Open the Hardware Setup dialog box.

Open the Allocation Editor dialog box.

Open the Mask List editor.

Open the Socket List editor.

Communication toolbar

Send the current project to the VCP.

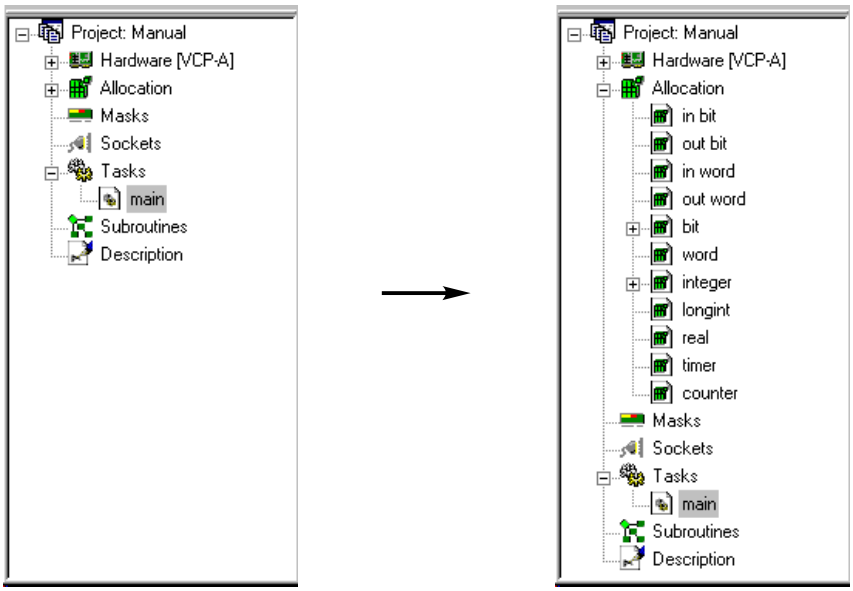
Open the on-line Variable Monitor.

Start VCP program.

Stop VCP program and turn off all outputs.

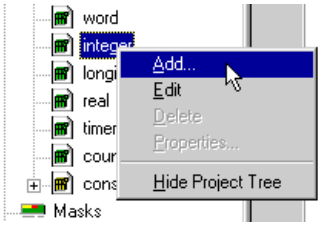
Project tree

Project tree displays all the project parts hierarchically. Left click on ⊞ will expand the tree, and allow a more detailed view. To shrink selection click ⊟.



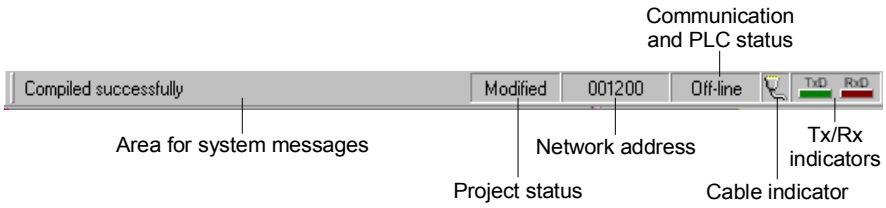
Right click on any component will open a context sensitive pop-up menu. Depending on type, it is possible to Add, Edit, Delete or change Properties of the selected component.

For example, to allocate a new integer variable click ⊞ besides Allocation to show all available variable types, right click integer and then choose Add.



Status bar

Status bar shows various information about the communication and the VCP. Default position of the status bar is at the bottom of the main window.



Project status is displayed if the current project is not saved. It reflects changes in either source, allocation, mask, socket or monitor list.

Network address shows the network address of the currently selected VCP. Right click to select another address or to enter a new address. The list of the network addresses will be saved together with the project.

Communication and VCP status can show:

Off-line	VCP is not responding. Check power supply, communication port, network address, communication cable and connections.
Run	VCP is on-line and running.
Stop	VCP is on-line, but is stopped. To run it, press the Start button.
Error	VCP is on-line, but some error occurred. To clear the error press Stop. Error codes are listed in the appendix.
Loader	VCP is on-line, but the system is corrupted. Start Kernel Maintenance and send a new kernel. Appropriate kernel will be supplied with each CyPro release.
Busy	PC is connected to the running network of VCPs, so it is impossible for the PC to communicate until the network master is stopped.

Cable indicator indicates if communication cable is properly connected:

Cable OK.
Cable check failed.

Tx/Rx indicators show communication activity. Tx indicates transmit (green) and Rx indicates receive (red).

Menu

File

New Project	Create a new empty project.
Open	Open an existing project.
Load From VCP	Load project from the VCP.
Save	Save the current project.
Save As	Save the current project under a new name.
Print	Print the current project.
Print Preview	Preview printed pages before printing.

Exit	Close current project and exit CyPro.

Edit

Undo	Cancel last action.
Redo	Cancel last Undo operation.
Cut	Delete the selection and put it on the clipboard.
Copy	Copy the selection onto the clipboard.
Paste	Insert text from the clipboard to the insertion point.
Delete	Delete the selection.
Select All	Select a whole document.
Find	Find the specified text.
Replace	Find the specified text and replace it.
Find Next	Find the next occurrence of the specified text.
Find Previous	Find the previous occurrence of the specified text.
Syntax Check	Check the current project for errors.

View

Hardware Setup	Open the Hardware Setup dialog box.
Allocation Editor	Open the Allocation Editor dialog box.
Mask Editor	Open the Mask List editor.
Socket Editor	Open the Socket List editor.
Subroutine List	Open the Subroutine List.

Insert

New Task	Create and open a new task.
New Subroutine	Create and open a new subroutine.

Communication

Get PLC info	Retrieve information about the VCP, check integrity of system and project memory, compare current project to one loaded into VCP.
Send To PLC	Send current project to the VCP.
Variable Monitor	Open on-line Variable Monitor.
Start PLC Program	Start a VCP program.
Stop PLC Program	Stop a VCP program and turn off all outputs.
Kernel maintenance	Download a new release of system software into VCP.

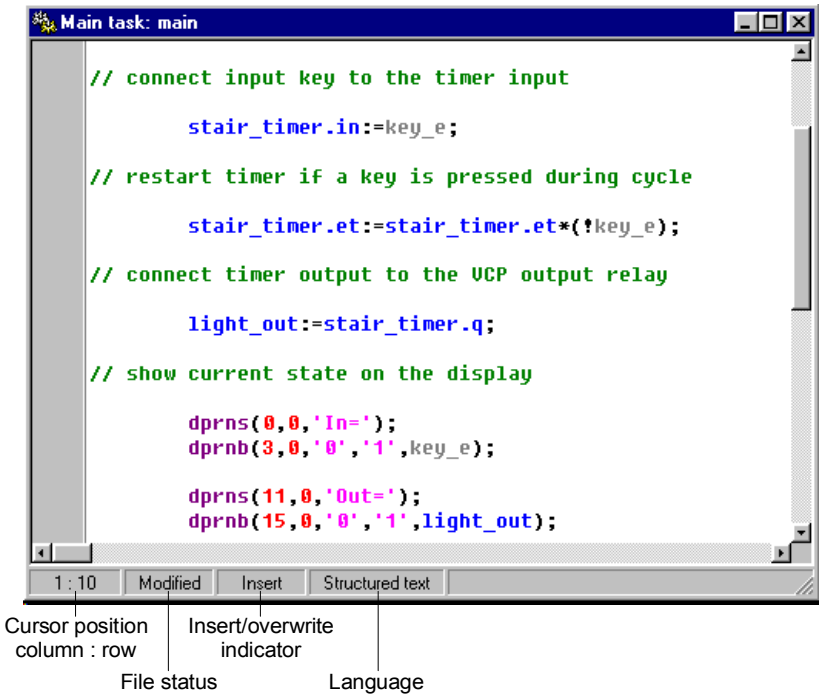
Options

Project	Change options related to the current project.
Environment	Change configuration options of the CyPro environment.

Edit window

Edit window is used to type and edit the VCP program. Each window contains one program module, that can be either task or subroutine. Each subroutine has to be opened in a separate window.

Instructions may be written in any available language, but different languages cannot be mixed in the same window. To change current language, select Edit/Properties.

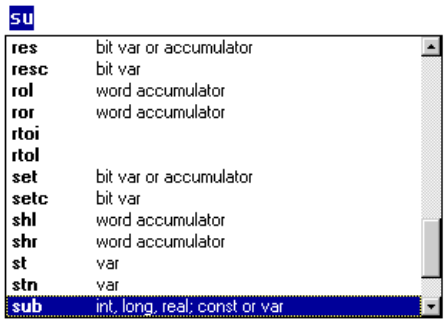


Editor content is dynamically syntax highlighted. Variables, constants, functions and other language elements are displayed in different colors. To select different color scheme or to customize colors, use Options/Environment/Colors.

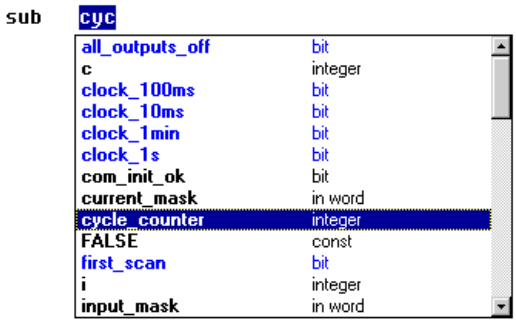
Edit instruction list

The instruction list language is usually entered in three columns, separated by tabs. The first column is usually empty, the second is for instructions, the third is for parameters and the fourth is for short comments. Longer comments usually start at the beginning of a line.

To start writing instruction list code, press **Tab** once. The insertion point will jump into the second column. Type in the instruction or, if you can't remember the exact name or spelling, press the **Insert** key. Small window will appear with the list of instructions and allowed arguments. Select the appropriate instruction and press **Enter**. It is also possible to type in a few letters and then bring up the list, or to press a letter after the list is opened.



Press **Tab** again. Insertion point wil jump to the third column. Type the first few letters of the variable and press **Insert**. Small window will appear again, but this time it will contain the list of the declared variables. The cursor is positioned according to the typed letters. If the instruction is jmp or cal, the list should contain labels or subroutines. To insert selected variable press **Enter**.



Edit structured text

Code helper is also useful in the structure text, but functionality is not exactly the same. The first **Insert** will display a list of allocated variables, and the second will display a list of available functions. Please note that some functions are available only as option.

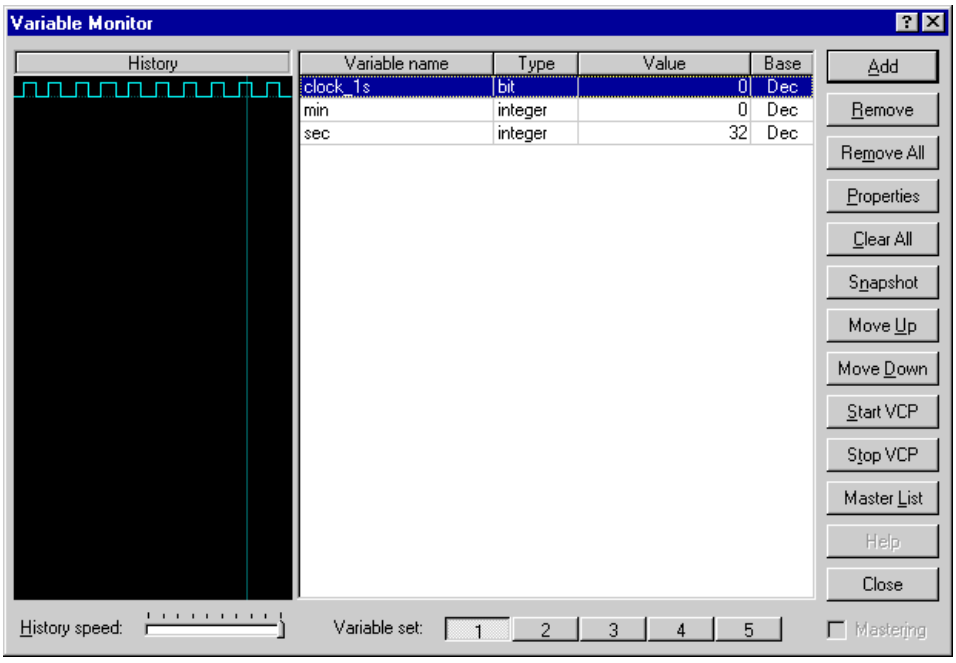
CyPro editor allows multiple level of undo operation. Text deleted or changed by mistake, can be restored by pressing **Ctrl-Z** key, even if other operations were made after. The number of undo commands depends on the available memory, but it is usually quite sufficient for any practical purpose.

CyPro also allows you to use an external editor. If project file is modified by external editor, compiler will offer option to reload project.

To examine structure of .cyp file open one of the enclosed examples.

Variable monitor

Variable monitor is a dialog box designed to display current values of VCP variables. To open it, select Communication/Variable Monitor, click Monitor icon on communication toolbar or press key **F10**.

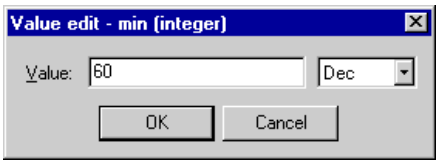


To insert variables click Add button or press **Insert** key. Dialog box Variable Insert will appear. Select desired variables and press OK. To select continuous block of variables, press **Shift** key. Multiple selections are possible when **Control** key is pressed. To rearrange list, select variable and click Move Up or Move Down, or press **Control** and move variable with **Up** or **Down** keys.

Variable monitor allows up to 5 monitored variables sets. To access a specific set, click buttons 1 to 5 or press keys **Alt-1** to **Alt-5**. To switch to the next or the previous set, press **PgUp** or **PgDn** keys.

Monitor values are updated approximately every 100ms. History scrolling speed can be changed using History speed slider.

To change variable value press the Edit Value button or press **Alt-Enter**.



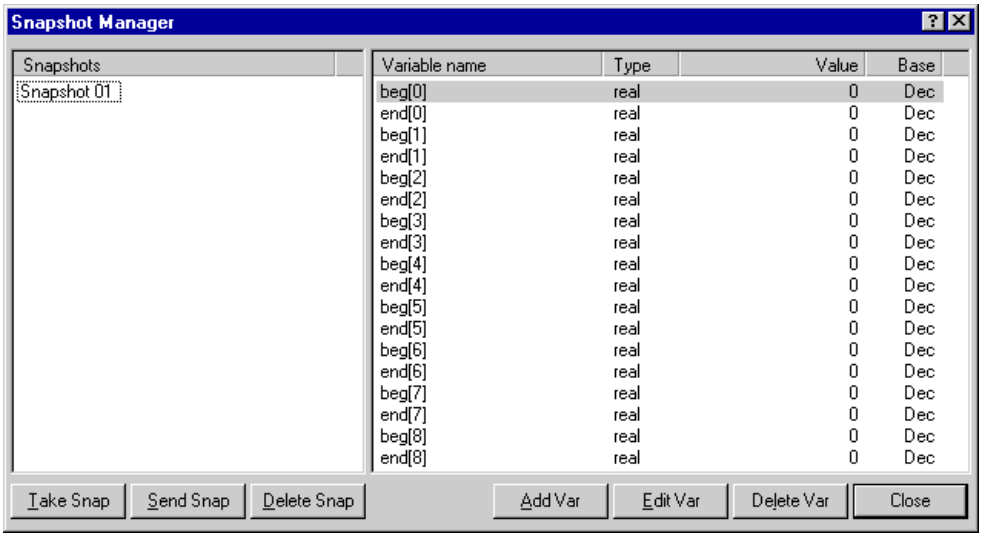
Enter new value and press OK. The entered value is sent to VCP and then read back, so the monitor always displays the actual value. Note that if VCP is running and the program writes in the same variable, its value will be rewritten in the next scan cycle.

To toggle value of a bit variable, press **Space**.

Snapshot

Snapshot contains values of the monitor variables taken at a particular moment. The snapshot can later be sent back to the VCP to reset variable values to the state when the snapshot was

taken. To take the snapshot, press the Snapshot button and press the Take Snap button in the Snapshot Manager. The Snapshot Manager will now contain a new snapshot item which will contain a set of variables along with their values.



Variable list can be modified by adding variables, editing values and deleting variables from the list.

To send snapshot to the VCP press Send Snap button. To delete snapshot from the project, press Delete Snap button. To rename selected snapshot, press **F2** key.

Programming

Hardware

IEX expansion modules

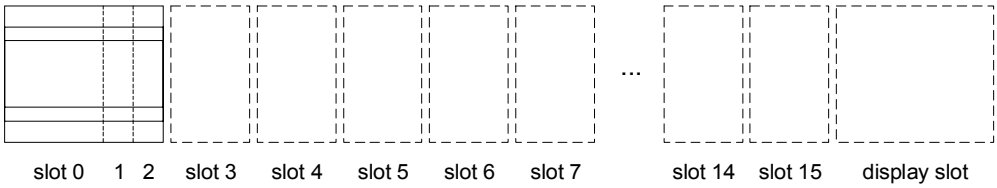
VCP can be expanded with various I/O modules, IEX cards and IEX expansion units.

IEX card

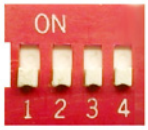
IEX expansion unit

Every IEX module occupies one “slot”. The slot is not a physical device, it is just a placeholder to put an expansion unit into.

Each slot has its unique slot address, numbered from 0 to 15. Slot 0 is occupied with local VCP inputs and outputs, so it should never be used by any IEX module. In addition, a dedicated display slot is available for default operator panel.



IEX cards determines slot addresses from their physical position. IEX expansion units have DIP switches that determines their slot addresses. If one or both internal CyBro slots are empty, their addresses may be used by IEX expansion units.

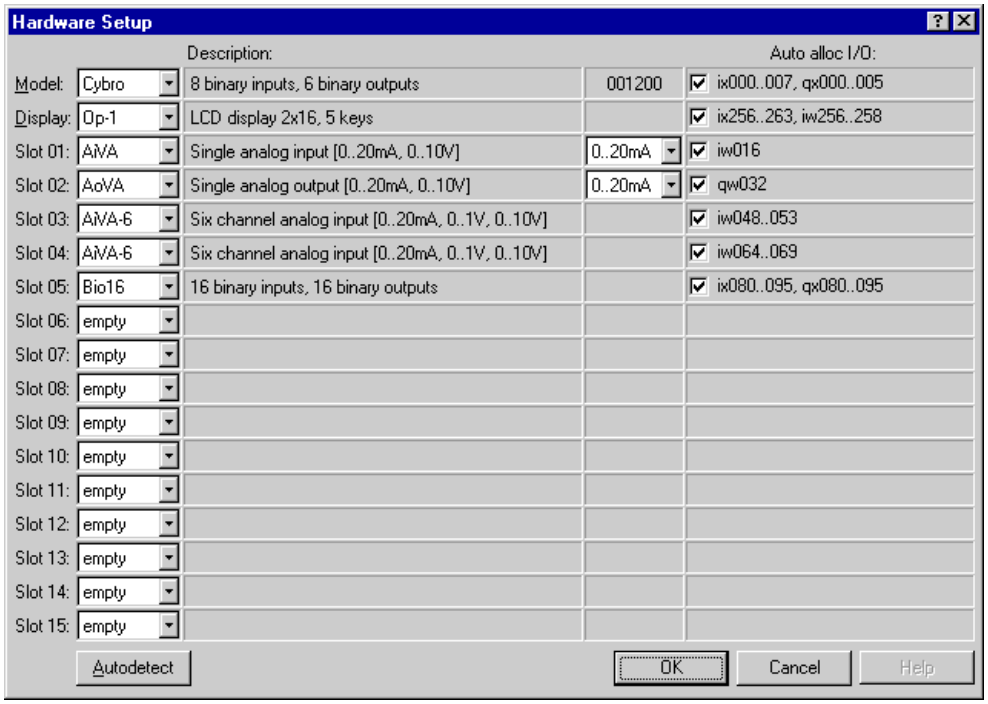


Some IEX modules have limited address range. To check available address range of a particular IEX unit, please look at the corresponding hardware manual.

Hardware setup

The first step of the new project development is setting up the VCP hardware configuration. Hardware may be configured later, but due to availability of auto allocation mechanism, it is advisable to do it at the beginning.

To perform automatic hardware detection press Autodetect button. If necessary, enter the network address of the corresponding VCP. After autodetection, configuration will appear in the Hardware Setup dialog box.


Hardware Setup
Description:
Auto alloc I/O:
Model: Cybro 8 binary inputs, 6 binary outputs 001200 ix000..007, qx000..005
Display: Op-1 LCD display 2x16, 5 keys ix256..263, iw256..258
Slot 01: AiVA Single analog input [0..20mA, 0..10V] 0..20mA iw016
Slot 02: AoVA Single analog output [0..20mA, 0..10V] 0..20mA qw032
Slot 03: AiVA-6 Six channel analog input [0..20mA, 0..1V, 0..10V] iw048..053
Slot 04: AiVA-6 Six channel analog input [0..20mA, 0..1V, 0..10V] iw064..069
Slot 05: Bio16 16 binary inputs, 16 binary outputs ix080..095, qx080..095
Slot 06: empty
Slot 07: empty
Slot 08: empty
Slot 09: empty
Slot 10: empty
Slot 11: empty
Slot 12: empty
Slot 13: empty
Slot 14: empty
Slot 15: empty
Autodetect
OK
Cancel
Help

Detected VCP model and cards are shown in the first column. Second column shows a brief description. Third column may contain optional parameters related to particular card, such as input or output type. Last column shows auto allocation settings and I/O addresses which a specific card occupies.

If Auto alloc I/O is checked, variables for accessing I/O will automatically be allocated. Variables will be allocated with default names, and may be changed later in the Allocation Editor. Unchecking auto allocation will deallocate I/O variables, but only if they are not changed manually.

If Auto alloc I/O is not checked, user should manually allocate variables in this area using Allocation Editor.

To identify network address of an unknown VCP enter zero as a network address, open the hardware setup and run Autodetect. Network address will appear in the first line, together with the detected IEX cards. Addressing with zero is not allowed if more than one VCP is connected.

Variables

Naming convention

Variable can be any string of letters, digits and underlines provided that the first character is not a digit. The maximum length is 24 characters. Names are not case sensitive, so Valve_2 and valve_2 are the same variable. Use of special national characters is not permitted.

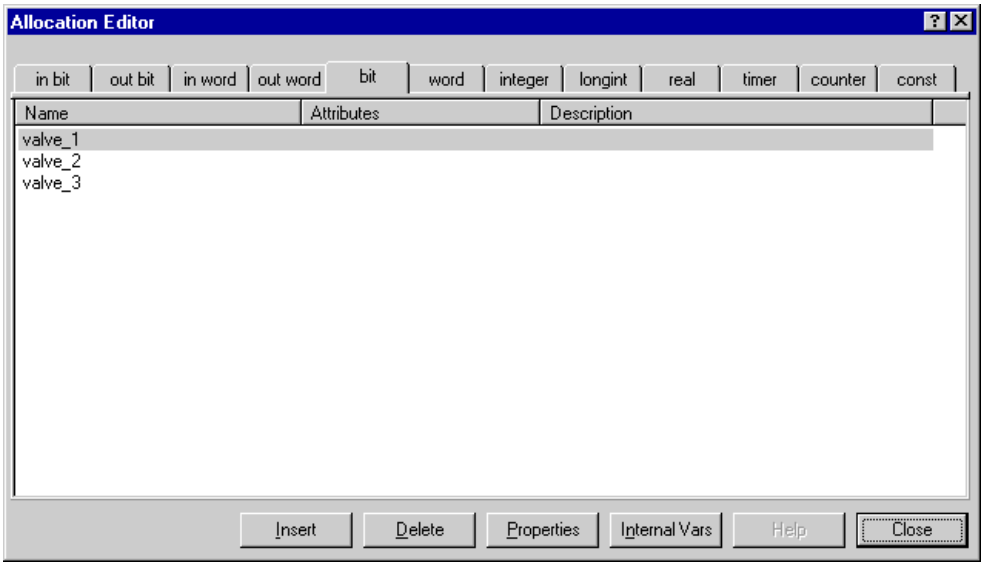
Examples of valid names are:

cnt
track5a
caret_position
valve_open_contact
MaximumWaterLevel

Allocation

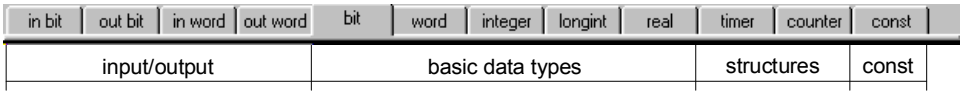
According to IEC-1131-3, memory cannot be accessed by address. Every variable should have a unique name and a strictly defined type. Same is valid for inputs and outputs.

Variables are allocated using Allocation Editor:



To insert a new variable select the appropriate type and press **Insert**. Enter name, select options and press **Enter**. If an incorrect name is entered, button OK is dimmed.

Data types may be grouped as follows:



In bit, out bit, in word and out word variables directly represents physically connected binary and analog signals.

Bit, word, integer, longint and real are basic data types.

Bit is a single boolean variable with only two states, zero or one. It should be used for flags, logical equations, logical states and similar. In bit and out bit are both bit type. The result of comparison instructions is also a bit type.

Word is a string of 16 bits. It doesn't represent a value, it is only a practical to perform logical operations on the 16 bits simultaneously. Arithmetic operations are not applicable for word variables.

Integer is a 16-bit signed number. It should be used for counting, encoding states, fixed point arithmetic and similar.

Longint is a 32-bit signed value. It should be used instead of integer when numbers higher then 32767 are expected.

Real is a floating decimal point number. It is used for arithmetic or when working with very large numbers. Real instructions typically consume three times as much memory as integer and processing real numbers is remarkably slower then integers.

Structures (timers and counters) are composed data types, consisting of several dedicated fields.

Constants are used to represent a value that will not be changed. As an example, Pi=3.14 can be defined for trigonometrical calculations. Data types do not apply for constants.

Data types used in VCP are strictly defined. Instructions to convert types are available, but should be used with caution in order to avoid errors. Limited number of type conversions indicates proper planning and good programming style.

Initialization

When program is started all non-retentive variables are initialized to zero.

To avoid initialization, declare variable as retentive. Retentive flag can be set for each variable individually, without restrictions. If needed, all variables can be retentive.

Sending a program to the controller will reset all variables, including the retentive ones. The only exception is sending a program where the allocation list is exactly matching one that already exists in the VCP. This usually happens when the user makes minor changes in the program. In such case, values of retentive variables are preserved.

Retentive variables retain their value even when the power is down. Data retention time is specified in the hardware manual.

I/O variables

Inputs and outputs cannot be accessed directly. Every input or output, binary or analog, should be declared before it is used.

VCP uses four address spaces for I/O access, two of them binary and two analog.

IX	0..255	input bit
QX	0..255	output bit
IW	0..255	input word
QW	0..255	output word

Binary inputs and outputs are allocated respectively, starting from the IX0 as the first physical input and QX0 as the first physical output. Remaining space is reserved for expansion cards.

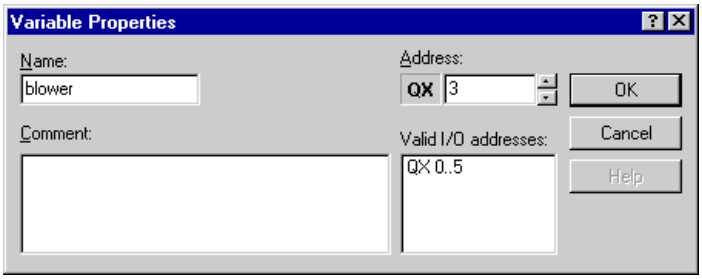
binary inputs

IX255 IX240	slot 15
. . .	
IX31 IX16	slot 1
IX15 IX0	on-board

binary outputs

QX255 QX240	slot 15
. . .	
QX31 QX16	slot 1
QX15 QX0	on-board

As an example, if blower is connected to output QX3, the proper declaration will be:



To turn the blower on manually, go to the Variable Monitor, add blower to the list and press **Space** key. It should work even if the VCP has stopped.

To turn the blower on from the structured text program, type:

```
blower:=1;
```

Send the program and start the VCP. The blower should run now.

Analog I/O cards have reserved 16 words for each slot. Single channel cards use only the first reserved word. Slot 0 is reserved, so the first available slot is 1. In word and out word variables are both integer data type.

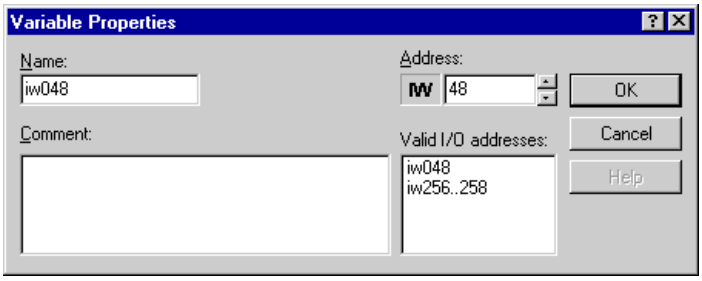
analog inputs

IW255 IW240	slot 15
. . .	
IW31 IW16	slot 1
IW15 IW0	reserved

analog outputs

QW255 QW240	slot 15
. . .	
QW31 QW16	slot 1
QW15 QW0	reserved

As an example, single Pt100 analog module inserted in third slot appears as IW48:



To check operation insert coil_temperature in the monitor list. With the resolution of 0,1°C and the actual temperature of 24,7°C, the monitor should display 247.

Input and output variables can be allocated by using auto allocation. By checking Auto alloc I/O checkbox in the Hardware Setup dialog, all variables related to expansion card are allocated/deallocated under the default name. To change name, open the Allocation Editor and press Properties button. Changed variables will not be automatically deallocated.

Internal variables

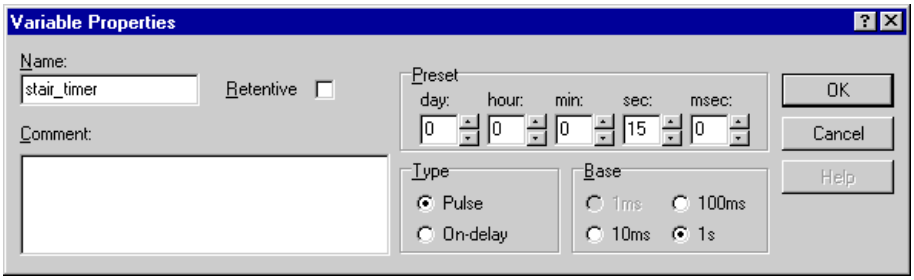
VCP provides several predefined variables. Each variable has a specific function.

clock_10ms	10ms system clock, 5ms high and 5ms low.
clock_100ms	100ms system clock, 50ms high and 50ms low.
clock_1s	One second system clock, 500ms high and 500ms low.
clock_1min	One minute system clock, 30s high and 30s low.
first_scan	Active during first scan after program run only. Useful to initialize variables.
scan_overrun	Indicates that scan timeout occurred. If a single program cycle lasts longer then 50ms, the program will be interrupted.
scan_time	Execution time of last scan in milliseconds.
scan_time_max	Maximal scan execution time encountered.
scan_frequency	Number of scan cycles per second. Zero if program stopped, more then 2000 for a short program. Also depends on number of I/O cards and I/O refresh priority.
io_refresh_time	Last update cycle for I/O cards in milliseconds.
io_refresh_time_max	Maximal I/O refresh time encountered.
retentive_fail	Indicates that the content of retentive memory is lost because the power was down for too long a period.
all_outputs_off	If active, all outputs will immediately go off.

To enable internal variables start Allocation Editor, press Internal Vars and mark all required variables. If particular variable is used in the VCP program it will be automatically enabled by the compiler.

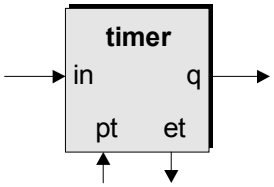
Timers

Timers are special variable types used to determine time interval. To define a new timer variable, open Allocation Editor, select timer type an press Insert.



Timer base is a period in which the timer is incremented. In other words, the base is the time resolution of the timer. The base should be equal or longer than the program scan time, so the 10ms base should not be used for a very long VCP program.

Timer may be represented as the function block with two inputs and two outputs:



Similarly, the timer variable consists of four fields. Each field is an elementary data type. Fields are:

name	direction	type	description
in	input	bit	input
q	output	bit	output
pt	input	long	preset time
et	output	long	elapsed time

To use timer from the PLC program, the following syntax applies:

```
<timer name>.<field>
```

For example, to set the preset of the wash_timer to 15 seconds (assuming that base is 100ms):

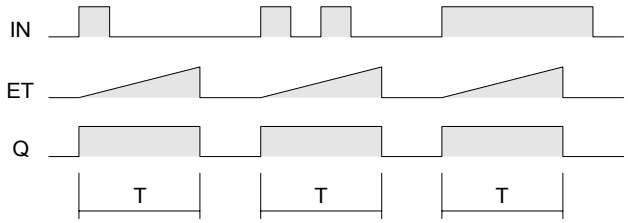
```
wash_timer.pt:=150;
```

Elapsed time of the wash_timer will start at 0 and increment every 100ms until it reaches 150.

Pulse timer

Timer output is activated immediately after the rising edge of input signal. After the certain specified time, the output will go off. Changes of input signal during pulse do not affect output.

The picture shows typical pulse timer operation:

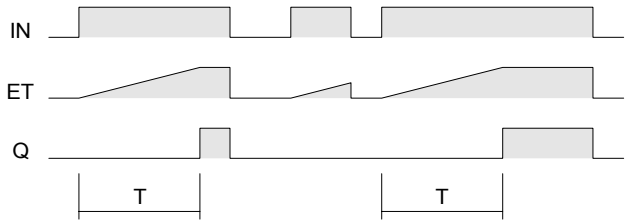


Typical application is staircase light timer.

On-delay timer

When input is activated, timer starts counting. After specified time output activates and stays high until input goes low. Available fields are the same as pulse timer.

The picture shows typical on-delay timer operation:

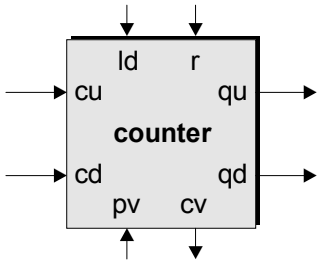


Typical application is star-delta switching for accelerating three-phase motors.

Counters

Rising edge on the counter input increments (or decrements) counter value by one. When value reaches preset limit, counter up output is activated. Every change is immediately visible, so output or value may be used from the next instruction on.

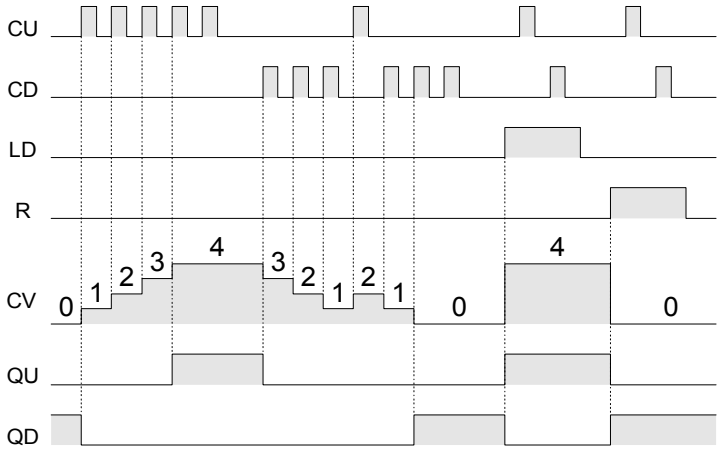
Counter may be represented as a function block with five inputs and three outputs:



Available counter fields are:

name	direction	type	description
cu	input	bit	count up
cd	input	bit	count down
ld	input	bit	load
r	input	bit	reset
pv	input	integer	preset value
cv	output	integer	count value
qu	output	bit	up limit
qd	output	bit	down limit

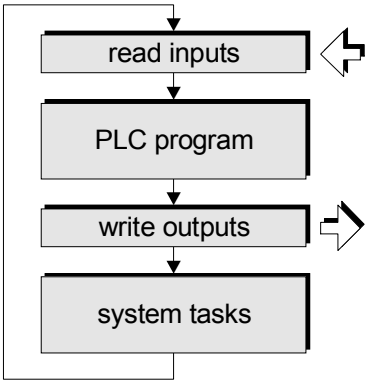
The picture shows a typical counter operation:



Preset value in the previous example is 4.

Program execution

VCP fully implements refresh processing system. It means that inputs are sampled immediately before, and outputs are refreshed immediately after the task execution. Even if input changes during task processing, input value remains stable during scan.



This is a little bit slower then direct processing, but the advantage is in much easier programming. Variable change in the middle of a scan is likely to introduce very nasty bugs when direct processing is implemented.

If, for any reason, scan time exceeds 50ms VCP declares scan overrun error and stops program. Error code is displayed on the status bar. If needed, checking for a scan overrun error may be disabled. To do this, uncheck Options/Project/General/Error checking.

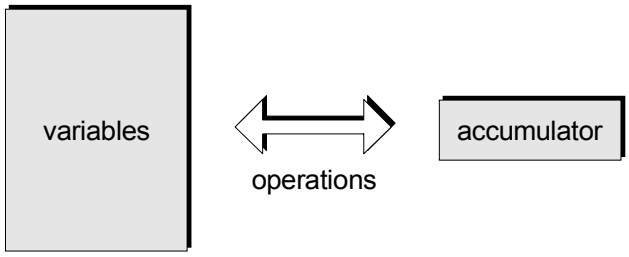
Instruction list

Instruction list is a low level language with structure similar to assembly language. Each line of instruction list code consists of four parts: label, instruction, operand and comment.

```
cnt_beg:    ld          product_count       // counting products
```

label instruction operand comment

Computation model of instruction list program consists of allocated variables and accumulator. All arithmetic and logic operations are performed on the accumulator.



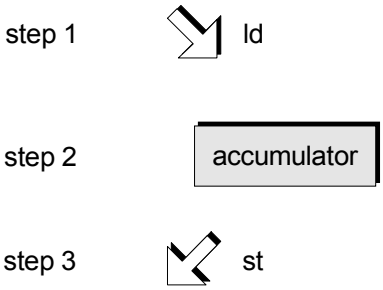
Unlike classical assembly language, accumulator may contain value of any type. Type is determined by the operand. If operand type is not unique, type is determined from subsequent instructions. Type tracking panel shows type for each line.

Once the accumulator is loaded, type cannot be changed until the result of the operation is stored. Exceptions are the comparison instructions and instructions for changing type.

Single operand instructions generally use accumulator as operand. In some cases, it is possible to execute the instruction directly on the variable.

The accumulator is always the first operand in the instructions with two operands. Second operand may be constant or variable. The result of the operation is also stored in accumulator.

Typical arithmetical or logical sequence should load value into accumulator (step 1), perform operation (step 2), and store result (step 3).



Each instruction should be written in a separate line.

For example, to increment variable product_count, the following sequence is needed:

```
ld          product_count
add         1
st          product_count
```

Value is loaded into accumulator, incremented and then stored back.

Instruction list allows two commenting styles. Single line comment starts with two slashes:

```
              ld            valve         // hot water
```

Multiline comment is bound within /* */ pair:

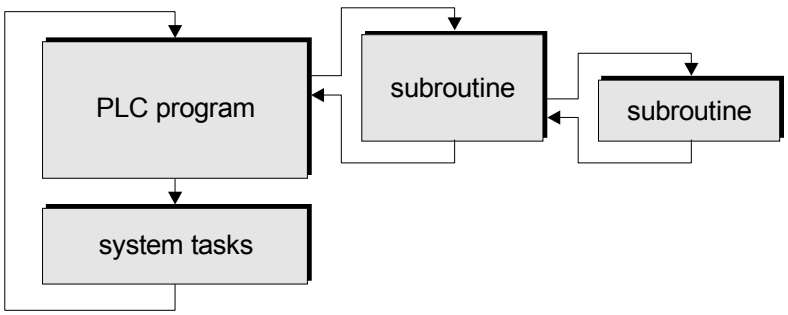
```
/*************************************************************
                      function move_caret()
*************************************************************
  input:        caret_position: integer
  output:       caret_done: bit
                caret_error_code: integer
  description: move caret until caret_position is reached
*************************************************************/
```

Labels are used to mark the destination of a jump instruction. Label should be written in the first column and should be followed by a colon:

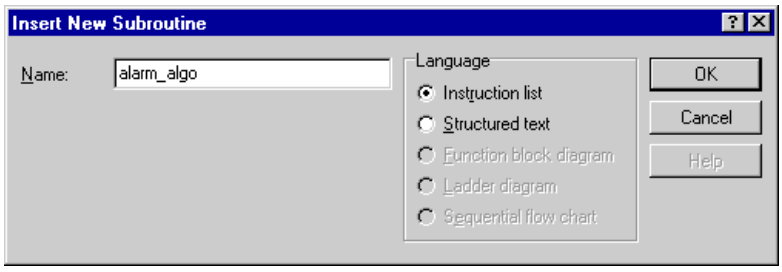
```
              jmp           block_end

              ld            test_value
              add           1
              st            test_value

block_end:
```

Jump instructions should be used carefully. Unconditional jump back may produce endless loop, which would cause scan overrun error.

Large programs may be split into subroutines. Subroutine contains a code segment that can be called from the main task or from another subroutine. Nesting level is limited only by the amount of available memory.



To create a new subroutine click Insert/New Subroutine, type name and press **Enter**.



Subroutine is called with the cal instruction:

```
              cal           alarm_algo
```

Subroutine can be called more times from different locations, but such subroutine should not contain fp or fn instructions. Subroutines may be written in different languages.

Load and store instructions

instruction:	LD
operation:	ACC := arg

bit	int	word	long	real	acc	const	var
•	•	•	•	•		•	•
change type to:					no change		

Loads constant value or variable value to accumulator.

instruction:	LDN
operation:	ACC := !arg

bit	int	word	long	real	acc	const	var
•						•	•
change type to:					no change		

Loads inverted bit variable value to accumulator.

instruction:	ST
operation:	var := ACC

bit	int	word	long	real	acc	const	var
•	•	•	•	•			•
change type to:					no change		

Loads accumulator value to specified variable.

instruction:	STN
operation:	var := !ACC

bit	int	word	long	real	acc	const	var
•							•
change type to:					no change		

Loads inverted accumulator value to specified variable.

instruction:	SET
operation:	arg := 1

bit	int	word	long	real	acc	const	var
•					•		•
change type to:					none, if var used		

Sets argument to logical TRUE condition.

instruction:	RES
operation:	arg := 0

bit	int	word	long	real	acc	const	var
•					•		•
change type to:					none, if var used		

Clears argument to logical FALSE condition.

instruction:	SETC
operation:	var := ACC | var

bit	int	word	long	real	acc	const	var
•							•
change type to:					no change		

Sets variable if accumulator is true, otherwise does nothing.

instruction:	RESC
operation:	var := !ACC & var

bit	int	word	long	real	acc	const	var
•							•
change type to:					no change		

Clears variable if accumulator is true, otherwise does nothing.

Logical instructions

instruction:	AND
operation:	ACC := ACC & arg

bit	int	word	long	real	acc	const	var
•		•				•	•
change type to:					no change		

Performs logical AND with accumulator and argument, and stores result to accumulator.

ACC before	arg	ACC after
0	0	0
0	1	0
1	0	0
1	1	1

instruction:	ANDN
operation:	ACC := ACC & !arg

bit	int	word	long	real	acc	const	var
•		•				•	•
change type to:						no change	

Performs logical AND with accumulator and inverted argument and stores result to accumulator.

ACC before	arg	ACC after
0	0	0
0	1	0
1	0	1
1	1	0

instruction:	OR
operation:	ACC := ACC | arg

bit	int	word	long	real	acc	const	var
•		•				•	•
change type to:						no change	

Performs logical OR with accumulator and argument, and stores result to accumulator.

ACC before	arg	ACC after
0	0	0
0	1	1
1	0	1
1	1	1

instruction:	OR
operation:	ACC := ACC | !arg

bit	int	word	long	real	acc	const	var
•		•				•	•
change type to:						no change	

Performs logical OR with accumulator and inverted argument, and stores result to accumulator.

ACC before	arg	ACC after
0	0	1
0	1	0
1	0	1
1	1	1

instruction:	XOR
operation:	ACC := ACC ^ arg

bit	int	word	long	real	acc	const	var
•		•				•	•
change type to:						no change	

Performs logical XOR with accumulator and argument, and stores result to accumulator.

ACC before	arg	ACC after
0	0	0
0	1	1
1	0	1
1	1	0

instruction:	XORN
operation:	ACC := ACC ^ !arg

bit	int	word	long	real	acc	const	var
•		•				•	•
change type to:					no change		

Performs logical XOR with accumulator and inverted argument, and stores result to accumulator.

ACC before	arg	ACC after
0	0	1
0	1	0
1	0	0
1	1	1

instruction:	SHL
operation:	-

bit	int	word	long	real	acc	const	var
		•			•		
change type to:					no change		

Performs logical bit shift to the left. Least significant bit (bit 0) is filled with 0.

accumulator contents before operation:

MSB LSB

1 0 0 0 1 0 0 0 0 0 1 1 1 0 0 0 ← 0

accumulator contents after operation:

MSB LSB

0 0 0 1 0 0 0 0 0 1 1 1 0 0 0 0

instruction:	SHR
operation:	-

bit	int	word	long	real	acc	const	var
		•			•		
change type to:					no change		

Performs logical bit shift to the right. Most significant bit (bit 15) is filled with 0.

accumulator contents before operation:

MSB LSB

0 → 1 0 0 0 1 0 0 0 0 0 1 1 1 0 0 0

accumulator contents after operation:

MSB LSB

0 1 0 0 0 1 0 0 0 0 0 1 1 1 0 0

instruction:	ROL
operation:	-

bit	int	word	long	real	acc	const	var
		•			•		
change type to:					no change		

Performs logical bit rotation to the left. Most significant bit (bit 15) is moved to the least significant bit (bit 0).

accumulator contents before operation:

MSB LSB

1 0 0 0 1 0 0 0 0 0 1 1 1 0 0 0 ←

accumulator contents after operation:

MSB LSB

0 0 0 1 0 0 0 0 0 1 1 1 0 0 0 1

instruction:	ROR
operation:	-

bit	int	word	long	real	acc	const	var
		•			•		
change type to:					no change		

Performs logical bit rotation to the right. Least significant bit (bit 0) is moved to the most significant bit (bit 15).

accumulator contents before operation:

MSB LSB

→ 1 0 0 0 1 0 0 0 0 0 1 1 1 0 0 0

accumulator contents after operation:

MSB LSB

0 1 0 0 0 1 0 0 0 0 0 1 1 1 0 0

instruction:	CPL
operation:	arg := !arg

bit	int	word	long	real	acc	const	var
•					•		•
change type to:					none, if var used		

Complements variable value.

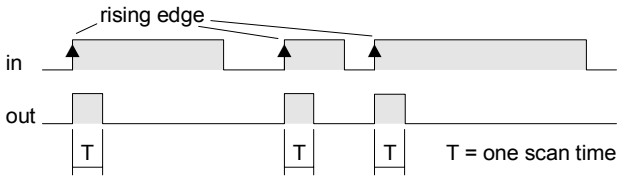
Edge instructions

instruction:	FP
operation:	-

bit	int	word	long	real	acc	const	var
•					•		•
change type to:					no change		

Detects the rising edge of accumulator value. When rising edge occurs, accumulator will contain TRUE condition.

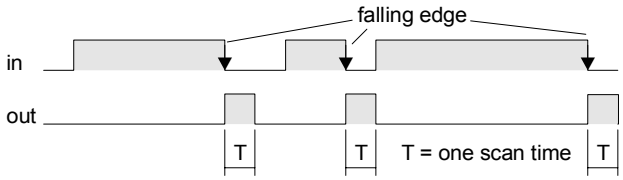
```
             fp          bit         // detect raising edge
             jmpnc       next        // jump over if edge isn't detected
             ld          cnt         // load variable cnt
             add         1           // increment by 1
             st          cnt         // store to cnt
next:
```



instruction:	FN
operation:	-

bit	int	word	long	real	acc	const	var
•					•		•
change type to:					no change		

Detects the falling edge of accumulator value. When falling edge occurs, accumulator will contain TRUE condition during one scan. The instruction to execute block of instructions can be used only once after the signal changes from 1 to 0.



Arithmetic instructions

instruction:	ADD
operation:	ACC := ACC + arg

bit	int	word	long	real	acc	const	var
	•		•	•		•	•
change type to:					no change		

Adds argument value to accumulator.

instruction:	SUB
operation:	ACC := ACC - arg

bit	int	word	long	real	acc	const	var
	•		•	•		•	•
change type to:					no change		

Subtract argument value from accumulator.

instruction:	MUL
operation:	ACC := ACC * arg

bit	int	word	long	real	acc	const	var
	•		•	•		•	•
change type to:					no change		

Multiplies accumulator value with argument and stores result to accumulator. No overflow detection is performed. If the result exceeds type boundaries, it will be truncated.

instruction:	DIV
operation:	ACC := ACC / arg

bit	int	word	long	real	acc	const	var
	•		•	•		•	•
change type to:					no change		

Divides accumulator value with argument and stores result to accumulator. If argument value is 0, illegal division is performed and result will be 0.

instruction:	MOD
operation:	ACC := ACC % arg

bit	int	word	long	real	acc	const	var
	•		•			•	•
change type to:					no change		

Calculates division reminder and stores it to accumulator. If argument value is 0, illegal division is performed and result will be 0.

instruction:	NEG
operation:	ACC := - ACC

bit	int	word	long	real	acc	const	var
	•		•	•	•		
change type to:					no change		

Negates accumulator value.

Comparison instructions

Comparison instructions are provided for comparing values of the same data type. All instructions return bit data type as result in the accumulator, TRUE if comparison succeeded and FALSE if it failed.

instruction:	EQ
operation:	ACC := ACC == arg

bit	int	word	long	real	acc	const	var
•	•	•	•	•		•	•
change type to:					bit		

Tests if accumulator is equal to argument, and stores result to accumulator.

instruction:	NE
operation:	ACC := ACC != arg

bit	int	word	long	real	acc	const	var
•	•	•	•	•		•	•
change type to:					bit		

Tests if accumulator is not equal to argument and stores result to accumulator.

instruction:	GT
operation:	ACC := ACC > arg

bit	int	word	long	real	acc	const	var
	•		•	•		•	•
change type to:					bit		

Tests if accumulator is greater than argument and stores result to accumulator.

instruction:	GE
operation:	ACC := ACC >= arg

bit	int	word	long	real	acc	const	var
	•		•	•		•	•
change type to:					bit		

Tests if accumulator is greater then or equal to argument and stores result to accumulator.

instruction:	LT
operation:	ACC := ACC < arg

bit	int	word	long	real	acc	const	var
	•		•	•		•	•
change type to:					bit		

Tests if accumulator is lesser than argument and stores result to accumulator.

instruction:	LE
operation:	ACC := ACC <= arg

bit	int	word	long	real	acc	const	var
	•		•	•		•	•
change type to:					bit		

Tests if accumulator is lesser tah or equal to argument and stores result to accumulator.

Type conversion instructions

Since data types used in VCP are imposed very strictly, there exists a number of type conversion instructions. Lower range types can be converted to higher range types without any data loss risk, but this isn't the case when converting higher to lower range type (e.g. long integer to integer). In the second case, the success of the conversion will depend on converting value.

instruction:	BTOI
operation:	-

bit	int	word	long	real	acc	const	var
•					•		
change type to:					integer		

Converts bit to integer.

instruction:	BTOL
operation:	-

bit	int	word	long	real	acc	const	var
•					•		
change type to:					long integer		

Converts bit to long integer.

instruction:	BTOW
operation:	-

bit	int	word	long	real	acc	const	var
•					•		
change type to:					word		

Converts bit to word.

instruction:	ITOL
operation:	-

bit	int	word	long	real	acc	const	var
	•				•		
change type to:					long integer		

Converts integer to long integer.

instruction:	ITOR
operation:	-

bit	int	word	long	real	acc	const	var
	•				•		
change type to:					real		

Converts integer to real.

instruction:	ITOW
operation:	-

bit	int	word	long	real	acc	const	var
	•				•		
change type to:					word		

Converts integer to word.

instruction:	LTOI
operation:	-

bit	int	word	long	real	acc	const	var
			•		•		
change type to:					integer		

Converts long integer to integer. Note that accumulator value can be truncated if it exceeds integer bounds.

instruction:	LTOR
operation:	-

bit	int	word	long	real	acc	const	var
			•		•		
change type to:					real		

Converts long integer to real.

instruction:	LTOW
operation:	-

bit	int	word	long	real	acc	const	var
			•		•		
change type to:					word		

Converts long integer to word. Note that accumulator value can be truncated if it exceeds word bounds.

instruction:	RTOI
operation:	-

bit	int	word	long	real	acc	const	var
				•	•		
change type to:					integer		

Converts and rounds real to integer. Note that accumulator value can be truncated if it exceeds integer bounds.

instruction:	RTOL
operation:	-

bit	int	word	long	real	acc	const	var
				•	•		
change type to:					long integer		

Converts and rounds real to long integer. Note that accumulator value can be truncated if it exceeds long integer bounds.

instruction:	WTOI
operation:	-

bit	int	word	long	real	acc	const	var
		•			•		
change type to:					integer		

Converts word to integer.

instruction:	WTOL
operation:	-

bit	int	word	long	real	acc	const	var
		•			•		
change type to:					long integer		

Converts word to long integer.

Branching instructions

Branching instructions are defining program execution flow. Conditional branching instructions are executed only if accumulator contains TRUE value, while the unconditional ones are executed in every scan. Jumping instructions should be used carefully because of potential endless loops which causes VCP device reset. Also, circular calls from subroutines can cause the same pitfall (e.g. sub1 calls sub2, which calls sub1 again).

instruction:	JMP label

bit	int	word	long	real	acc	const	var
change type to:					none		

Unconditionally jumps to label.

instruction:	JMPC label

bit	int	word	long	real	acc	const	var
•							
change type to:					none		

Jumps conditionally to label if accumulator value is set to TRUE.

instruction:	JMPNC label

bit	int	word	long	real	acc	const	var
•							
change type to:					none		

Jumps conditionally to label if accumulator value is set to FALSE. Use JMPC and JMPNC instructions to branch after comparison.

instruction:	CAL subroutine

bit	int	word	long	real	acc	const	var
change type to:					none		

Unconditionally calls subroutine. After finishing subroutine execution, the program continues program execution after the CAL instruction.

instruction:	CALC subroutine

bit	int	word	long	real	acc	const	var
•							
change type to:					none		

Calls subroutine if accumulator value is TRUE.

instruction:	CALNC subroutine

bit	int	word	long	real	acc	const	var
•							
change type to:					none		

Calls subroutine if accumulator value is FALSE. Use CALC and CALNC instructions to conditionally call subroutines after comparison.

Structured text

Structured text is a high level language with a syntax somewhat similar to Pascal, but it is specifically developed for industrial control applications.

Assignment statements

Assignment statements are used to store value in variable. An assignment statement has the following general format:

```
variable := expression;
```

The assigned value should be an equal or lower data type as the variable.

Expressions

Expressions are used to calculate values derived from other variables and constants. Expression always produces a value of a particular data type. An expression may involve one or more constants, variables, operators or functions. Using expressions, VCP can perform complex arithmetic computations involving nested parenthesis and/or different data types.

Examples:

```
y_position:=5;
down_timer.pt:=15000;
circle_area:=r*r*3.14;
case_counter:=case_counter+1;
volts:=amps*ohms;
start:=(oil_press and steam and pump) and not emergency_stop;
valid_value:=(value = 0) or ((value > 10) and (value <= 60));
```

Operators

VCP utilizes a number of arithmetic and logical operators, listed in the following table:

operator	alias	unary	binary	bit	word	int	long	real	result type
+			•			•	•	•	same
-		•	•			•	•	•	same
*			•			•	•	•	same
/			•			•	•	•	same
mod	%		•			•	•		same
not	!	•		•	•				same
and	&		•	•	•				same
or	|		•	•	•				same
xor			•	•	•				same
=	==		•	•	•	•	•	•	bit
<>	!=		•	•	•	•	•	•	bit
<			•			•	•	•	bit
<=			•			•	•	•	bit
>			•			•	•	•	bit
>=			•			•	•	•	bit
:=			•	•	•	•	•	•	same

Operators are sorted by precedence. Some operators have alias which may be used instead of standard mnemonic.

Expression evaluation

Expressions are evaluated in a particular order depending on precedence of the operators and other sub-expressions. Parenthesized expressions have the highest precedence. Operators of the highest precedence are evaluated first, followed by lower precedence operators, down to the lowest. Operators of the same precedence are evaluated left to right.

Consider following example:

```
Speed1 := 50.0;
Speed2 := 60.0;
Press  := 30.0;
Rate   := Speed1/10 + Speed2/20 - (Press + 24)/9;
```

Evaluation order is:

```
Speed1/10  = 5
Speed2/20  = 3
Press+24   = 54
54/9       = 6
5+3        = 8
8-6        = 2
Rate       = 2
```

To change evaluation order add brackets:

```
Rate:=Speed1/10+Speed2/(20-(Press+24)/9);
```

The expression (20-(Press+24)/9) has higher precedence and will be evaluated before its value is used as a divisor for Speed2.

The value for Rate in this case will be:

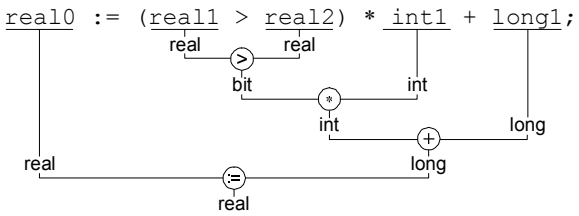
```
      = 5 + 60/(20 - 6)
      = 5 + 60/14
Rate = 9.286
```

Data type conversions

Type conversions are performed automatically, but only lower-to-higher type conversions are valid:

bit → integer → word → longint → real

In the following example, a number of conversions will be performed.



Multiline expressions

It is possible to write a multiline expression, but every line must end with an operator. Following example shows how expression could be broken into several lines.

```
heater_on := (heater_temperature < 600) and
             (((mode = MANUAL) and start_pressed) or
              ((mode = AUTO) and heater_on_request)) and
             not emergency_stop;
```

Conditional statements

Conditional statements are provided so that selected statements could be executed if a certain condition exist.

if...then...else

Block of statements can be evaluated and executed depending on the value returned by a boolean expression using if...then construction.

This takes the general form:

```
if <boolean expression> then
  <statements>;
end_if;
```

The boolean expression can be any expression that returns a TRUE or FALSE boolean result, e.g. the state of a single bit variable or a complex expression involving numerous variables.

Alternative statements can be executed using the if...then...else construction in the form:

```
if <boolean expression> then
  <statements>;
elsif <boolean expression> then
  <statements>;
else
  <statements>;
end_if;
```

Examples of conditional execution:

```
if collision then
  speed:=0;
  brakes:=ON;
end_if;

if (gate=closed) and (pump=on) and (temp>200) then
  control_state:=active;
else
  control_state:=hold;
  pump_speed:=10;
end_if;
```

If...then and if...then...else constructions can be nested within other conditional statements to create more complex conditional statements.

```
if flow_rate>230 then
  if flame_size>4 then
    fuel:=4000;
  else
    fuel:=2000;
  end_if;
else
  fuel:=1000;
end_if;
```

Further statements can be conditionally executed within the if...then using the elsif construction, which has the general form:

```
if <boolean expression> then
  <statements>
elsif <boolean expression> then
  <statements>
else
  <statements>
end_if;
```

Any number of additional elsif sections can be added to the if...then construction.

```
if a>b then
  d:=1;
elsif a=b+2 then
  d:=2;
elsif a=b-3 then
  d:=4;
else
  d:=3;
end_if;
```

case...of

The case conditional statement is provided so that selected statements can be executed depending on the value of an expression that returns a boolean, integer or long integer result, e.g. the value of a single variable or the value from the complex expression.

The set of statements which have a constant value that matches the value of the expression are executed. If no match is found, the statements preceded by else will be executed.

The case construction has the following general form:

```
case <expression> of
  <value1>: <statements>;
  <value2>: <statements>;
  <value3>: <statements>;
else
  <statements>;
end_case;
```

Example:

```
case material_type of
  1: speed:=5;
  2: speed:=20;
  3: speed:=25;
     fan:=ON;
  4: speed:=30;
     fan:=ON;
  5: speed:=50;
     fan:=ON;
     water:=ON;
else
  speed:=0;
end_case;

case alarm_bit of
  TRUE:  speed:=0;
  FALSE: speed:=MAX_SPEED;
end_case;
```

Iteration statements

Iteration statements are provided for situations where it is necessary to repeat one or more statements a number of times, depending on the state of the particular variable or condition.

Iteration statements should be used carefully in order to avoid endless loops, which will cause scan overrun error.

for...do

The for...do construction allows a set of statements to be repeated depending on the value of an iteration variable. This is an integer or long integer variable which is used to count the statements executions. Iteration variable is incremented by 1 at the end of for...do loop.

This construction takes the general form:

```
for <var>:=<expression> to <expression> do
  <statements>;
end_for;
```

The test to check whether the iteration variable value has reached the final value is always made before executing the statements. After leaving for...do construction, iteration value will contain the final expression value.

The statements within a for...do construction should not modify variables that will affect the expressions for the final and increment values.

Example:

```
for i:=0 to 19 do
  channel[i]:=TRUE;
end_for;

for t:=lo_value-1 to hi_value*2 do
  tank_number:=t;
  state[t]:=t/2;
end_for;
```

while...do

The while...do construction allows one or more statements to be executed while a particular boolean expression remains true. The boolean expression is tested prior to executing the statements. If it is false, the statements within the while...do will not be executed.

This construction takes the general form:

```
while <expression> do
  <statements>;
end_while;
```

Example:

```
while value<(max_value-10) do
  value:=value+position;
end_while;
```

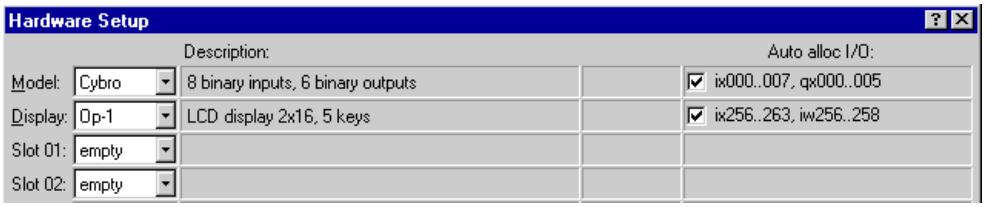
While...do loop is very rare in a typical PLC program.

Operator panel

General

Operator panel (OP) is the optional external device connected to the VCP via the IEX bus. OP provides LCD display and a few keys readable from the PLC program.

For a proper operation, OP has to be defined in the Hardware Setup dialog box. Hardware setup is saved together with project.



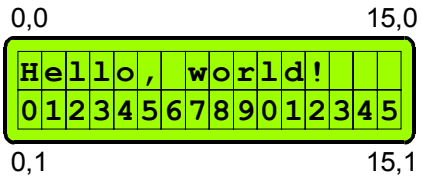
To program operator panel, the following tools are available:

Print functions	Structured text functions typed in the PLC program. Used to display strings and values.
OP keys	Bit variables readable from PLC program. Represents operator panel keys.
Masks	Visual tool for programming operator panel, used to enter parameters. Capable of entering integer values, decimal values and values represented by strings. Parameters may be hierarchically organized.

Print functions

Print functions are structured text functions, used to display text messages and values.

First two parameters of all functions are x and y coordinates. They are used to set display position. Print origin is in the upper left corner. Printing outside the visible display range is not allowed and may produce unexpectable results.



Print functions are:

```
dclr();
```

Display clear.

Clears the whole display. Requires no parameters.

```
dprnc(x:int, y:int, c:char);
```

Display print ASCII character.

Prints single character on the specified coordinates. String may contain any characters from the extended ASCII set (character codes 32 to 255).

```
dprns(x:int, y:int, str:string);
```

Display print string.

Prints character string on the specified coordinates. The string is an array of characters enclosed in the single quotes. It may contain any characters from the extended ASCII set (character codes 32 to 255).

```
dprnb(x:int, y:int, c0:char, c1:char, value:bit);
```

Display print binary value.

Prints first or second ASCII character on the specified coordinates, depending on the bit value. If the value is false the first character is printed, otherwise the second.

```
dprni(x:int, y:int, w:int, zb:bit, value:int);
```

Display print integer value.

Prints integer value to the specified coordinates. Parameter w defines printing width. For example, if w is 4, valid print range is from -999 to 9999. Parameter zb is binary and stands for a zero blanking. If z is 1 leading zeroes are not printed.

```
dprnl(x:int, y:int, w:int, zb:bit, value:long);
```

Display print long value.

Prints long value to the specified coordinates. Parameter w defines printing width. For example, if w is 6, valid print range is from -99999 to 999999. Parameter zb is binary and stands for a zero blanking. If zb is 1 leading zeroes are not printed.

```
dprnr(x:int, y:int, w:int, dec:int, value:real);
```

Display print real value.

Prints real value to the specified coordinates. Parameter w defines printing width, and parameter dec defines number of printed decimals. For example, if w is 6 and dec is 2, valid print range is from -99.99 to 999.99. Zero blanking is always on.

Every function parameter (except string for dprns) may be a constant, a variable or an expression. This may be used to create animated displays, as in the following example:

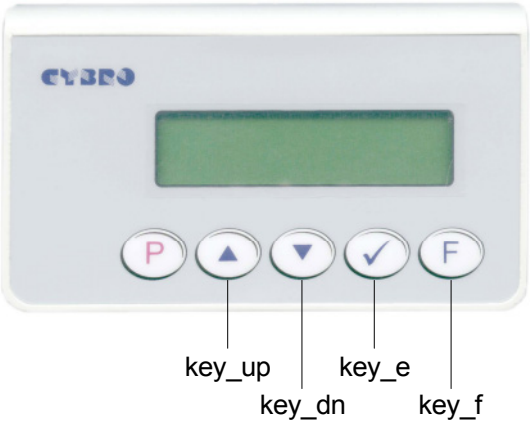
```
  dclr();
  dprns(0,0,'Moving...');
  dprns(x,1,'o');
  x:=(x+fp(clock_100ms))%16;
```

It is possible to connect more than one operator panel to the VCP. In that case, each display function should have additional parameter, slot number. This parameter is entered first, before the x coordinate, and represents the slot number of the desired display.

OP keys

OP keys are accessible from the PLC program as the input bit variables.

Available keys:



The **P** key is reserved and it is not accessible from the PLC program. Its function is to invoke mask system. When the mask is active, the keys **up**, **dn** and **E** are also reserved, so their value in PLC program is zero.

The keys do not have the autorepeat function. The key variable is true as long as the key is pressed. After the key is released, key variable becomes false.

Any two (or more) keys may be pressed simultaneously. This may be used to initiate some destructive or rarely used operations. In the following example, pressing **up** and **dn** simultaneously will reset variable product_count.

```
if fp(key_up and key_dn) then
  product_count:=0;
end_if;
```

If Auto aloc I/O in the Hardware Setup dialog box is checked, key variables will be automatically allocated.

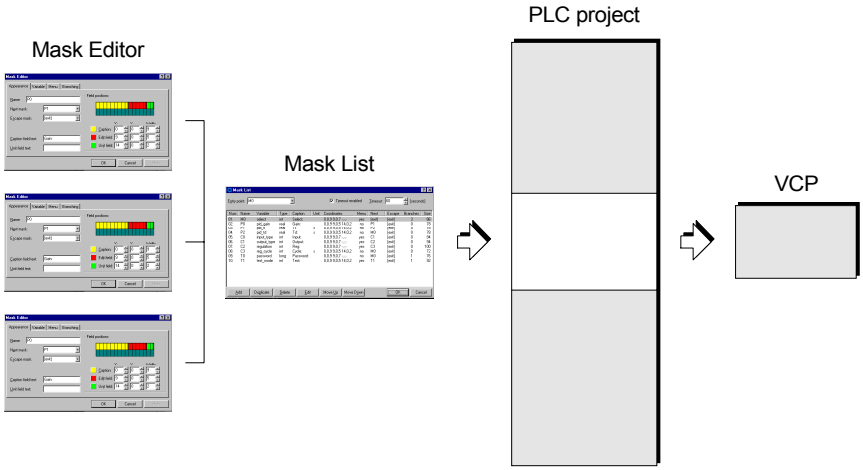
Display and keys are updated approximately every 100ms. It depends on the number of IEX modules, length of PLC program and I/O refresh priority. To check the refresh time make visible internal variable io_refresh_time (Allocation Editor/Internal Vars) and add it to Variable Monitor.

Masks

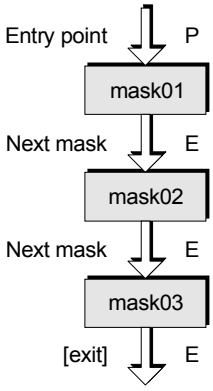
Mask system is easy-to-use visual tool for programming operator terminal.

Mask is a container for a variable that will be edited. Masks are transferred to the VCP together with VCP code.

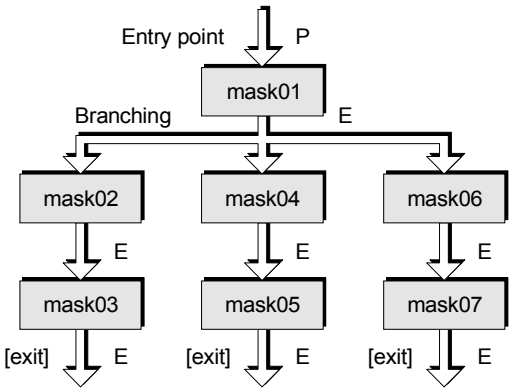
User creates a new mask or edits the existing one by using the Mask Editor. Created masks are listed in the Mask List. Masks are integral part of the PLC project, they are saved on the disc and transferred to the VCP.



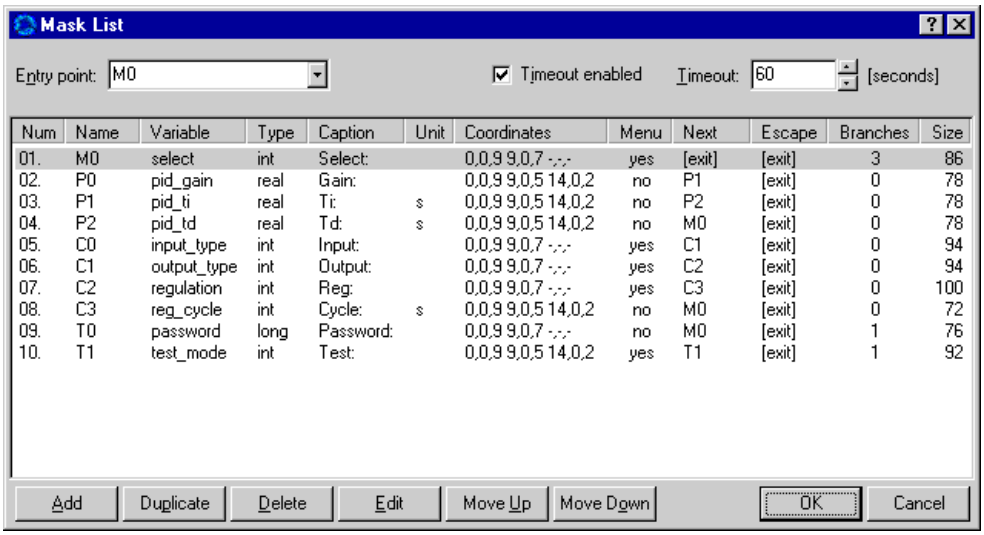
When user presses **P** on the operator terminal, VCP sends first mask to the OP. Pressing **E** advances it to the next mask.



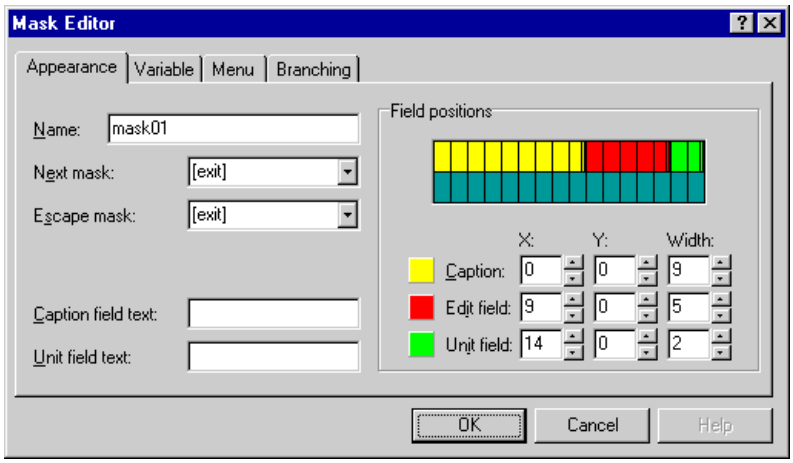
By use of “branching” feature, masks may be hierarchically organized:



To begin working with masks, press Masks button on the standard toolbar or press key **F7**. Mask List dialog box will appear.


Mask List
Entry point: M0
Timeout enabled
Timeout: 60
[seconds]
Num Name Variable Type Caption Unit Coordinates Menu Next Escape Branches Size
01. M0 select int Select: 0,0,9 9,0,7 -,-,- yes [exit] [exit] 3 86
02. P0 pid_gain real Gain: 0,0,9 9,0,5 14,0,2 no P1 [exit] 0 78
03. P1 pid_ti real Ti: s 0,0,9 9,0,5 14,0,2 no P2 [exit] 0 78
04. P2 pid_td real Td: s 0,0,9 9,0,5 14,0,2 no M0 [exit] 0 78
05. C0 input_type int Input: 0,0,9 9,0,7 -,-,- yes C1 [exit] 0 94
06. C1 output_type int Output: 0,0,9 9,0,7 -,-,- yes C2 [exit] 0 94
07. C2 regulation int Reg: 0,0,9 9,0,7 -,-,- yes C3 [exit] 0 100
08. C3 reg_cycle int Cycle: s 0,0,9 9,0,5 14,0,2 no M0 [exit] 0 72
09. T0 password long Password: 0,0,9 9,0,7 -,-,- no M0 [exit] 1 76
10. T1 test_mode int Test: 0,0,9 9,0,5 14,0,2 yes T1 [exit] 1 92
Add
Duplicate
Delete
Edit
Move Up
Move Down
OK
Cancel

To create a new mask click Add or press **Insert** key. Mask Editor dialog box will appear.



Name is a unique string identifier which identifies a particular mask.

Next mask defines a mask which becomes active after **E** key is pressed. In other words, this is a mask that contains the next variable.

Escape mask defines a mask which becomes active after **P** key is pressed. Usually, this is used to exit from the mask.

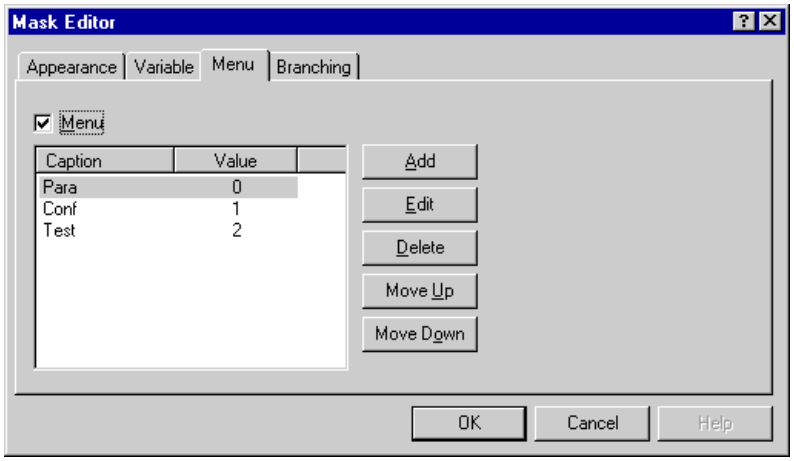
Caption field is a short string that will appear on the display to identify the currently edited variable. Caption position is represented by the yellow rectangle. To move the caption, drag the rectangle into the desired position. To resize caption, drag the right edge of the rectangle.

Edit field is a display area in which the value of edited variable is displayed. It is represented by the red rectangle. Edit field should have enough space for editing variable in the desired range. To move and resize field, drag it like the caption.

Unit field is a short string, similar to caption. Unit field is represented by the green rectangle, and it is usually used for displaying engineering units.

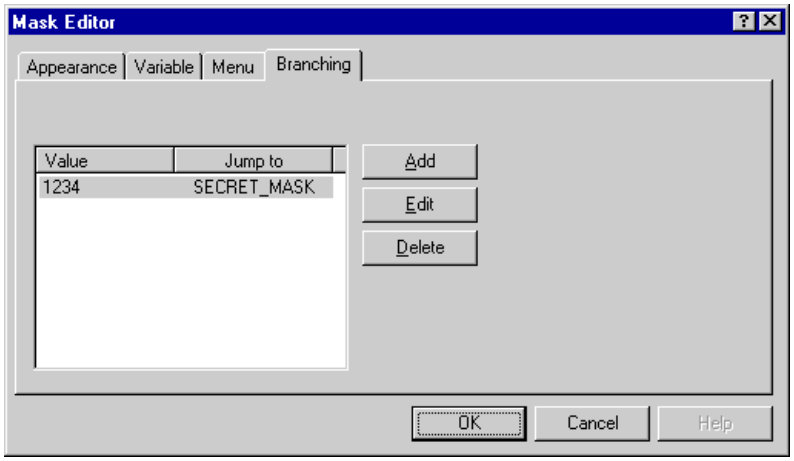
Field may be hidden by adjusting its width to zero.

Some variables may be entered as menu rather than as numerical value. To define menu entries, run Mask Editor, click Menu tab and Add as many items as needed.



When executing VCP program, the display will show items by name, and variable product_type will take value 0, 1 or 2.

Branching tab provides branching onto different masks according to the entered value. This can be used to organize parameters into various parameter sets, but also for a password protected parameters.



Active mask takes control of all panel keys except the **F** key, so it is not possible to use them from VCP program at the same time. Mask fields are displayed “over” the user display. After exiting mask, display content is restored.

If mask is too large to fit into operator panel it will not be activated, and it will operate like an empty mask. Mask size is displayed in Mask List dialog box. Available operator panel mask memory is displayed in the Hardware Setup dialog box. To decrease mask size reduce number of menu entries or reduce edit field width. Reducing caption and unit field width may also save few bytes.

It is not possible to activate more masks simultaneously.

VCP program can access information about currently active mask by reading variable current_mask. When current_mask is zero no mask is active. Program can also force execution of the certain mask by writing to the variable input_mask.

Small example program MaskArray.cyp ilustrates principle of controlling masks from the VCP program.

Com2 port

General

Com2 is a serial communication port available for communication with ultrasonic sensors, scales, serial printers and others.

Communication protocol is defined by the PLC program. Binary message format is possible, but included communication functions are most suitable for sending, receiving and parsing plain ASCII messages. Both master and slave operation is supported. Com2 port is full duplex, so characters may flow simultaneously in both directions.

Com2 port is optional, so it should be specified by the VCP order code. To check if com2 port of currently connected VCP is enabled start Communication/Get PLC Info/Kernel.

Initialization

Com2 port should be initialized before any other communication command is applied. To initialize port, use function:

```
com_init(baud_rate:long, data_bits:int, parity:int, stop_bits:int):bit;
```

Parameter *baud_rate* determines communication speed, expressed as a number of bits per second (bps). Common baud rates are:

1200
2400
4800
9600
19200
38400

Other available baud rates may be calculated as follows:

$$baud\_rate = \frac{384000}{N}$$, where N is any whole number between 1 and 65536

Example: Required baud rate is 300bps. To check if it is obtainable, divide 384000 by 300. Result is 1280, without residual, so port may be initialized to 300bps.

Example: Required baud rate is 56kbps. 384000 divided by 56000 is 6.85, so that 56kbps is not available. The closest possible rate is 384000/7=54857, which is 2% off the required speed. In a certain application that may be acceptable, but it would be more appropriate to select another baud rate.

Maximum communication speed is 384kbps.

Parameter *data_bits* may be 7 or 8.

Parameter *parity* may be one of the following:

0 - no parity
1 - odd parity
2 - even parity

Parameter *stop_bits* may be 1 or 2.

Function com_init() returns true if initialization is successful, and false if not. Initialization may fail if VCP has no Com2 port enabled or if initialization parameters are out of range.

Common initialization sequence is:

```
if first_scan then
  com_open:=com_init(19200,8,0,1);
end_if;
```

This example initializes Com2 port to 19200bps, 8 data bits, no parity and 1 stop bit. Variable com_open may be used later to check if initialization was successful.

Prepare transmit message

Prior to transmission, PLC program should create complete outgoing message. To create a message use display functions dprnc(), dprns(), dprnb(), dprni(), dprnl() and dprnr().

To print to the transmit buffer, zero should be added before other parameters. Added parameter is in fact a slot number, so transmit buffer will appear as operator panel in the slot 0.

The X coordinate is a transmit buffer position. The Y coordinate should be zero. Maximum message length is 256 bytes.

For example, to write message “Hello!” enter:

```
dprns(0,0,0,'Hello!');
```

Message length is 6 characters.

Special characters may be entered as a two-character combination. The first character being a backslash ('\'), and second being a one of the following:

combination	ASCII code	hex code
\n	CR LF	0D 0A
\r	CR	0D
\t	TAB	09
\\	\	5C
\nn	any	nn

The last three-character combination may be used to enter a hexadecimal code of any ASCII character. For example, '\41' is equivalent to letter 'A'.

For example, to create the message “Hello!” followed by carriage return and line feed, enter:

```
dprns(0,0,0,'Hello!\n');
```

Message length is now 8 characters.

To create message composed of keywords and numerical values use:

```
  dprns(0,0,0,'>LEVEL=xxx.xx TEMPERATURE=xx.x ERROR=xx\n');
  dprnr(0,7,0,6,2,water_level);
  dprnr(0,26,0,4,2,water_temperature);
  dprni(0,37,0,2,no,error_code);
```

Note that x-es in the first line will be overwritten by the subsequent commands.

Transmit

To start transmitting use function tx_start():

```
tx_start(char_num:int);
```

Parameter *char_num* is the number of characters to transmit. Transmission always starts from beginning of the buffer. It is not possible to start transmitting from any other position.

```
tx_active():bit;
```

Function tx_active() returns current transmission state. When transmission is finished tx_active() drops to zero.

```
tx_count():int;
```

Function tx_count() returns the number of characters left to transmit. If tx_count() is zero, but tx_active() is still true, the last character is currently transmitted.

```
tx_stop();
```

If transmission is still in progress, and a certain condition requires the transmitter to stop immediately, function tx_stop() may be used.

Currently transmitted character will be transmitted completely, but the other characters will not.

Receive

Receiving is completely independent of transmitting.

Maximum length of a received message is 256 bytes. If more than 256 characters are received, receiving position rolls-over to the beginning of the receive buffer. In that case, the number of received characters is set to zero.

To start receiving use function rx_start(). This function also defines criteria to stop receiving:

```
rx_start(beg_ch:char, end_ch:char, len:int, msg_tout:int, char_tout:int);
```

Parameter *begch* specifies the first character of received message. After the receiving is initiated, any other character will be rejected until begch is received.

To receive a message without specifying the first character, put zero instead of begch.

Parameter *endch* specifies the last character of a received message. After endch is received, reception is stopped.

To receive a message without specifying the last character, put zero instead of endch.

Parameter *len* specifies the length of a received message. After the required number of bytes, the reception is stopped.

To receive a message of unspecified length, set len to zero.

Parameter *msg_tout* specifies message timeout in milliseconds. This is the time after which the receiver will quit if no characters are received. Maximum timeout is 32 seconds.

To receive without limiting message time, set msg_tout to zero.

Parameter *char_tout* specifies the receiving timeout between individual characters. If characters sent by accompanying device are sent continuosly without gaps, character timeout may be set to a pretty low value in order to improve the receiver response time. Character timeout should always be greater than the time needed to transmit a single character, concerning selected baud rate and data bits.

For example, if communication parameters are 1200 bps, 8 bits and no parity, transmission of one character will be approximately 8ms (start bit + 8 data bits + stop bit = 10bits; 10bits/1200bps=8.3ms). Character timeout should be no less then 10ms, although 50..100ms is more appropriate if the response is not time-critical.

To receive without character timeout set char_tout to zero.

Both timeouts are fully independent of each other. It is possible to set character timeout, and leave message timeout disabled. The opposite is allowed but meaningless.

A few examples will illustrate usage of the rx_start() function:

Receive characters continuously from the serial port:

```
rx_start(0,0,0,0,0);
```

Receive message beginning with '>' and ending with CR character:

```
rx_start('>','\r',0,0,0);
```

Receive message exactly 12 characters long:

```
rx_start(0,0,12,0,0);
```

Infinitely wait for a first character, but stop receiving 250ms after message is received:

```
rx_start(0,0,0,0,250);
```

Stop receiving 100ms after the last received character or if no character is received in 5 seconds:

```
rx_start(0,0,0,5000,100);
```

Any combination of start and stop criteria is allowed. To receive message up to 80 characters, ending with Ctrl-Z (1A hex) with 10 seconds timeout:

```
rx_start(0,'\1A',80,10000,100);
```

If the given criteria for stopping receiver are not appropriate for a requested communication protocol, the determining of the end of the message is left to the PLC program. Received message should be analyzed on the fly, and when stop condition is satisfied, function rx_stop() may be used to stop receiving:

```
rx_stop();
```

Function rx_count() returns the number of received characters. The function may be used whether the receiving is active or not. Function rx_start() resets this number to zero.

```
rx_count():int;
```

To check if the receiving is active, use function:

```
rx_active():bit;
```

To check detailed receiving status, use function:

```
rx_status():int;
```

Function rx_status() returns one of the following codes:

0 - receiving active
1 - function rx_stop() executed
2 - end character detected
3 - requested number of characters received
4 - timeout

Example: Incoming message has no terminating character, and the length is variable. Message length is coded binary in the fourth byte. To receive such messages use the following code:

```
  if rx_active() and rx_count()>=4 then
    if rx_count()>=rx_bufrd(3) then
      rx_stop();
    end_if;
  end_if;
```

Parse received message

After receiving, PLC program should analyze the received message. Subroutine for this task usually has to:

- check message format and decide if message is acceptable or not
- convert received data into PLC variables

If message format is strictly declared the analyzing procedure may be simple, but it is much better to write a fairly flexible program capable of handling format changes. In an extreme but not unlikely example, the whole received message may be rejected by an inflexible parse procedure because a single extra space is added somewhere.

Functions for parsing a received message:

```
rx_bufrd(position:int):int;
```

Returns single character from the given position of the receive buffer. Character is converted to integer value in range 0..255.

```
rx_strcmp(position:int, str:string):bit;
```

Compares the receive buffer with a specified string. If the string matches, return value is true otherwise it is false.

```
rx_strpos(position:int, str:string):int;
```

Searches for a specified string in the receive buffer. The search starts from a given position. If the string is found, function returns position of the first matching character; otherwise, return value is -1.

```
rx_strtoi(position:int):int;
```

Returns integer value of decimal number starting from the given position. If a character at the specified position is space, next character is taken until number is found. Conversion continues until the first non-digit character is encountered.

```
rx_strtol(position:int):long;
```

Returns long value of decimal number starting from the given position. If character at a specified position is space, the next character is taken until the number is found. Conversion continues until the first non-digit character is encountered.

```
rx_strtor(position:int):real;
```

Returns real value of decimal number starting from the given position. If character at a specified position is space, the next character is taken until the number is found. Conversion continues until the first non-digit character is encountered.

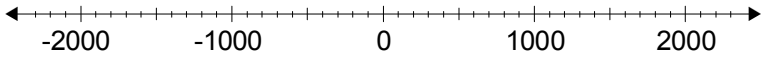
Example: Communication is used to set the parameters of the current PLC program. Message contains keywords or pairs 'NAME=VALUE', separated by spaces. Each parameter has its unique name. Incoming message may contain requests to set one or more parameters.

```
  if rx_strpos(0,'OPEN')<>-1 then
    main_valve=1;
  end_if;

  if rx_strpos(0,'CLOSE')<>-1 then
    main_valve=0;
  end_if;

  position=rx_strpos(0,'TEMP=');
  if position<>-1 then
    set_point=rx_strtoi(position+5);
  end_if;

  position=rx_strpos(0,'CYCLE=');
  if position<>-1 then
    cycle_timer.pt=rx_strtol(position+6);
  end_if;
```

High speed counter

General

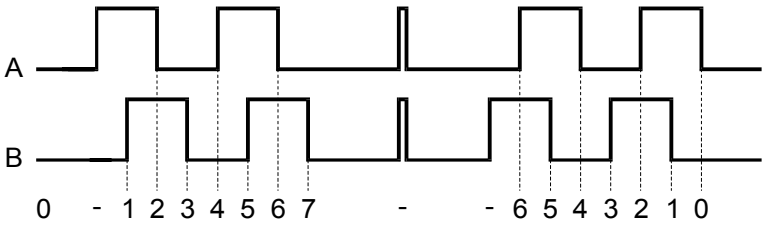
High speed counter (HSC) is a hardware device capable of counting external pulses from an encoder in both directions. The counter is a 32-bit signed register, so that the counting range is practically unlimited.



Counter inputs are two phase-shifted signals usually called A and B.

Unlike most other counters, counting is performed on every detected edge, so the effective resolution is quadrupled. Encoder, declared at 500 pulses per rotation, will increment counter by 2000 in every rotation cycle.

Other characteristics are glitch removal and ±1 pulse removal. This feature eliminates improper counting when encoder shaft vibrates.



High speed counter is optional, so it should be specified by the VCP order code.

Two variations are possible: AB or AB+Z. Second type has additional zero input, usually used to determine absolute position of the controlled device.

High speed counter and COM2 port are mutually exclusive, so it is not possible to have them together on the same VCP. Also, HSC with zero input (AB+Z) restricts COM1 port to RS232 only, so it is necessary to use an external RS232/RS485 converter to connect such VCP to the network.

Table below summarizes possible HSC variations:

	high speed counter	zero input	COM1-232	COM1-485	COM2
none	-	-	+	+	+
AB	+	-	+	+	-
AB+Z	+	+	+	-	-

After the power-up, the value of the high speed counter is undetermined.

High speed inputs may also be used as standard binary inputs.

For connections and technical specifications of HSC, please check the hardware manual.

High speed counting

By default, the high speed counter is stopped. To start counting use function hsc_start(). To stop counting use function hsc_stop(). To determine if HSC is stopped or started, use function hsc_active(). If HSC is started hsc_active() returns true, otherwise returns false.

To read the current position of high speed counter use function hsc_read(). Reading is possible even if the counter is currently active and running. Counter value type is longint, and may be stored in a variable of the same type.

```
current_position:=hsc_read();
```

To set the counter to the desired value use function hsc_write(). Usually this is performed only when the counter is stopped, but it is possible to write a new value even if the counter is active and running.

```
hsc_write(12000);
```

As an example, the following line increments the counter by 500 counts.

```
hsc_write(hsc_read()+500);
```

High speed action

High speed counting is usually used for a precise motion control. The following example shows the way stop the motor that is connected to qx000 output, when the counter reaches zero (downward counting).

```
if hsc_read()<=0 then
  qx000:=0;
end_if;
```

Problem that might arise here is the unpredictable response time. If current scan time is 5ms, response time may be anywhere between 0ms and 10ms. For a speed of 1m/s that induces inaccuracy of 10 millimeters.

To prevent this, high speed counter is equipped with a hardware capable of performing a very fast action when counter reaches zero. Action is defined as setting or resetting a single binary output. To perform the same task as in the last example, execute the following line:

```
hsc_set_action(0,qx000);
```

The function hsc_set_action() initiates action. When counter reaches zero, the output qx000 will be deactivated. Also, it is possible to set action in the opposite direction, to activate the output. See example:

```
hsc_set_action(1,brake_out);
```

When counter reaches zero, brakes will be activated.

Action should be initiated only once, so the typical code for activating action is:

```
if fp(start_key) then
  hsc_write(300000);
  hsc_set_action(0,qx000);
  hsc_start();
  qx000:=1; // start motor
end_if;
```

Function hsc_reset_action() is provided to cancel initiated action. This is useful when something unexpected happens, for example when alarm condition is detected.

```
if alarm_condition then
  hsc_reset_action();
  hsc_stop();
  qx000:=0;
  qx001:=0;
end_if;
```

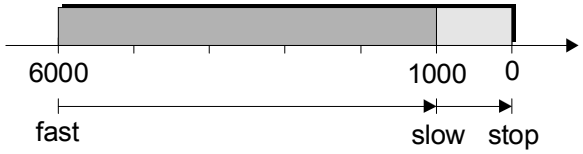
Action is performed only once, when counter reaches zero for the first time. When counter reaches zero again, no action will be performed. To check if the action is still pending, use function hsc_check_action(). If action is initiated, hsc_check_action() returns true. After the action is performed (or canceled), hsc_check_action() returns false.

It is not possible to initiate more then one action simultaneously. An attempt to do so will cause incorrect operation. If multiple actions are needed, next action may be initiated after the first action is performed. In applications that need multi-speed control, gear change may also be performed by if..then instructions, rather than actions. Usually, this allows enough precision to slow down, with a single high speed action used to stop accurately.

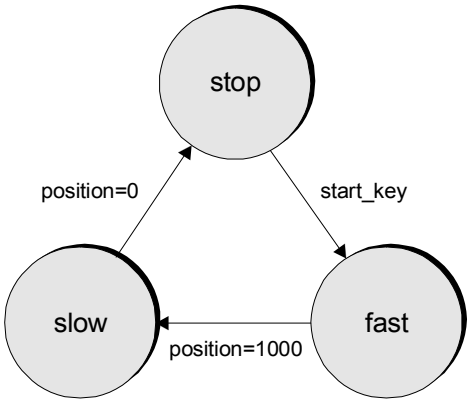
Next example shows a typical two-speed application. Two-speed motor is activated by two binary outputs, qx000 and qx001. Low speed is activated by qx000, and high speed is activated by qx001.

	stop	slow	fast	fast
qx000	0	1	0	1
qx001	0	0	1	1

The example requires motion of 6000 units. For the first 5000 units motor will run fast, then for the next 1000 units it will run slowly, and then it will stop. Slowing down will allow accurate stop position.



Program consists of three main states: stop, fast and slow. Transitions between states are defined by start_key and position.



The actual code may be written on both first or second manner, by the if..then instruction or by the two consecutive actions.

The first way is somewhat shorter and easier to understand:

```
if qx000=0 and qx001=0 then
// currently stopped
  if fp(start_key) then
    // start full speed and set action to stop after 6000
    hsc_write(6000);
    hsc_set_action(0,qx000);
    hsc_start();
    qx000:=1;
    qx001:=1;
  end_if;
elsif qx000=1 and qx001=1 then
// currently fast
  if hsc_read()<=1000 then
    // slow down
    qx001:=0;
  end_if;
end_if;
```

The same functionality may be performed in another way, by the two sequentially activated actions. The code is little longer and less understandable, but both actions will be performed very accurately.

```
if qx000=0 and qx001=0 then
// currently stopped
  if fp(start_key) then
    // start full speed and set action to slow down after 5000
    hsc_write(5000);
    hsc_set_action(0,qx001);
    hsc_start();
    qx000:=1;
    qx001:=1;
  end_if;
elsif qx000=1 and qx001=0 then
// currently slow
  if not hsc_check_action() then
    // continue slow, increment counter by 1000 and stop on zero
    hsc_write(hsc_read()+1000);
    hsc_set_action(0,qx000);
  end_if;
end_if;
```

Zero reset

Third VCP type (AB+Z) has additional high speed input, used to accurately determine absolute position.

Usual function of zero input is to reset high speed counter. By default this is disabled, and may be enabled by the function hsc_enable_zero(). If enabled, each inactive-to-active transition on the zero input will reset high speed counter to zero.

To disable zero reset, use function hsc_disable zero(). To check if the zero reset is enabled, use function hsc_check_zero(). If the reset is enabled, hsc_check_zero() will return true, otherwise it will return false.

Zero reset is not a single-shot function, so a once activated reset is active until it is explicitly deactivated by the hsc_disable_zero() function.

If zero reset is enabled and action is activated, action will be performed immediately after the transition is detected.

Zero detect

The second way of utilizing zero input is provided by the two additional functions, hsc_detect_zero() and hsc_read_zero().

Function hsc_detect_zero() indicates transition on the zero input. If inactive-to-active transition on the zero input was detected, hsc_detect_zero() will return true, but only the first time. The next consecutive calls will return false, until the next zero transition is detected.

That allows simple transition counting:

```
  zero_counter:=zero_counter + hsc_detect_zero();
```

When transition is detected the value of the high speed counter is also written to the zero detect register, readable by the function hsc_read_zero(). Zero detect register is also 32 bits long.

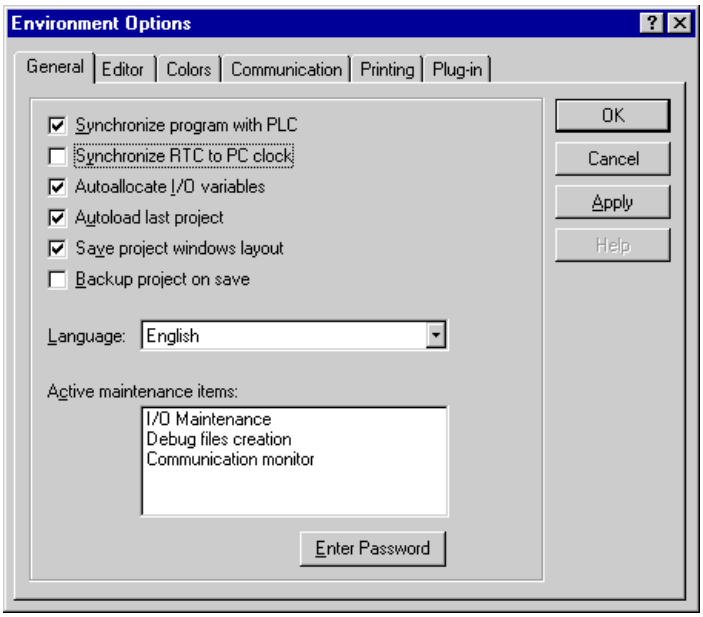
Zero detection functions are usable on the AB+Z VCP type only.

Real-time clock

General

Real-time clock (RTC) is a hardware clock/calendar device. It runs even when the power supply is down. For technical specifications about accuracy and data retention time, please check the hardware manual.

RTC is adjusted according to PC clock every time the PLC program is transferred to the PLC. To prevent this, clear the check box Options/Environment/General/Synchronize RTC to PC.



RTC can also be adjusted from the PLC program with rtc_write functions.

RTC read

To read current time use functions:

```
rtc_read_hour():int;
rtc_read_min():int;
rtc_read_sec():int;
```

Return values are in the range:

hour	0..23
min	0..59
sec	0..59

To read current date use functions:

```
rtc_read_year():int;
rtc_read_month():int;
rtc_read_date():int;
```

Return values are in the range:

year	2000..2099
month	1..12
date	1..31

To read current day of the week use function:

```
rtc_read_weekday():int;
```

Function return value is coded as follows:

0 - Sunday
1 - Monday
2 - Tuesday
3 - Wednesday
4 - Thursday
5 - Friday
6 - Saturday

Rtc_read functions may be used without restrictions.

RTC adjust

To set time use functions:

```
rtc_write_hour(hour:int);
rtc_write_min(min:int);
rtc_write_sec(sec:int);
```

To set date use functions:

```
rtc_write_year(year:int);
rtc_write_month(month:int);
rtc_write_date(date:int);
```

To set day of the week use function:

```
rtc_write_weekday(weekday:int);
```

In contrast to read, rtc_write functions should be used with caution. When time or date is adjusted all functions should be called in one scan cycle, because RTC cannot be stopped.

An example that displays date/time on the operator panel and allows user to adjust them is RtcDemo.cyp. The program is located in the Project\Examples directory.

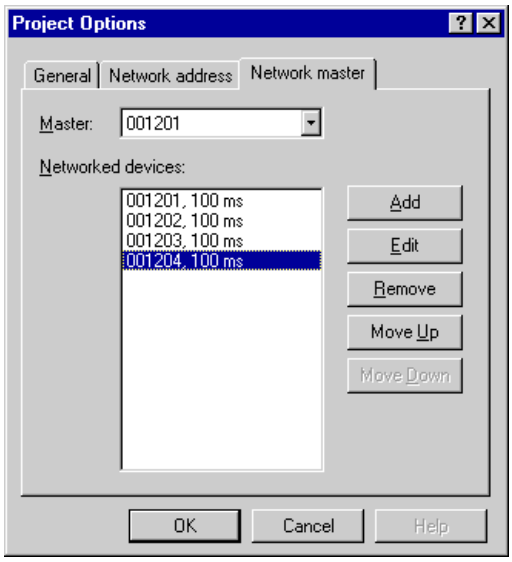
Networking

General

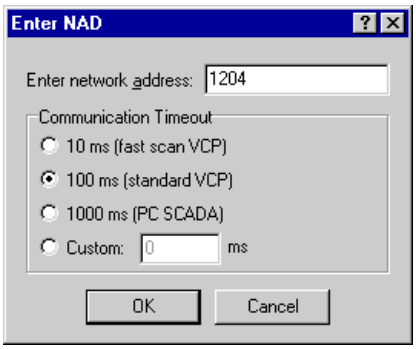
VCP can be used as a stand-alone device, as well as a part of the A-bus network. Network provides easy and efficient data exchange between a number of devices. Specified variables will be distributed through the network and may be used by any networked device. Every A-bus device has its own unique network address (NAD) which identifies it on the network.

Network should contain one (and only one) network master. All other VCPs are slaves. The master initiates all network communication and controls data exchange. To create a master, open Options/Project Options/Network master and select NAD of desired VCP. Any VCP may be a network master.

The master should have a list of network devices. More precisely, only devices that have to send data (and have at least one output socket) should be listed. Devices that only receive data may be omitted from the list of networked devices.



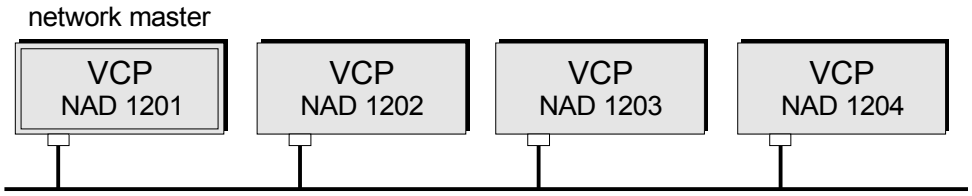
To add a new device to the list, press Add button. A small dialog box will appear. Enter NAD and press OK.



In most cases the appropriate communication timeout value is 100ms. Only when the related VCP has a very short scan time, communication timeout may be shorter, but it cannot be below its scan time plus a few milliseconds. PC-oriented SCADA systems may need a timeout value of a few hundred milliseconds or even more.

Master-slave relation has nothing in common with the data exchange direction. The master only determines sequence and rhythm of network messages. Every VCP in the network can send and receive data, including the master itself.

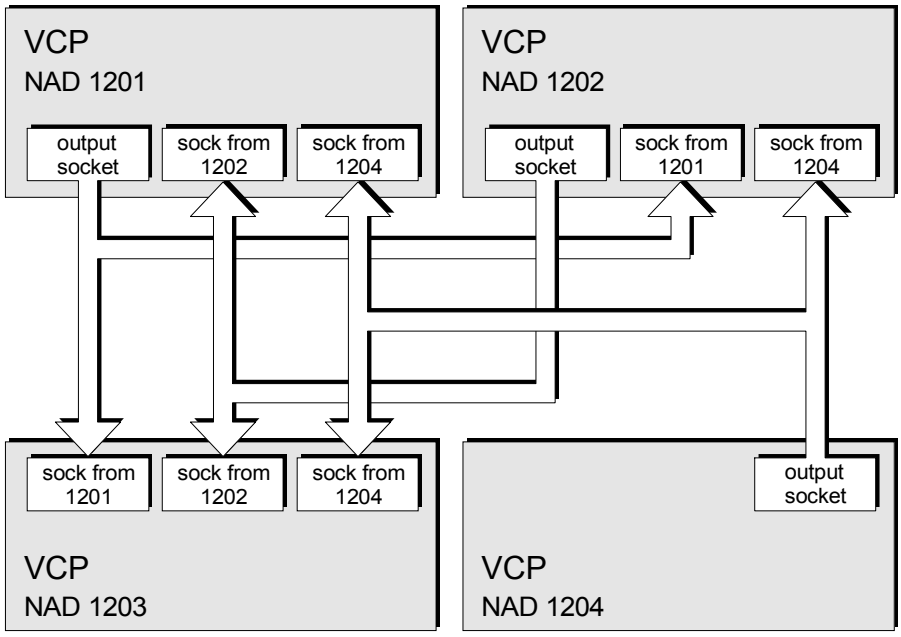
Network starts when the master is started by the hardware run/stop switch. Starting VCP from CyPro does not start network mastering. Similarly, master can only be stopped by the run/stop switch.



In this example, network master is the VCP with network address 1201, networked together with 1202, 1203 and 1204.

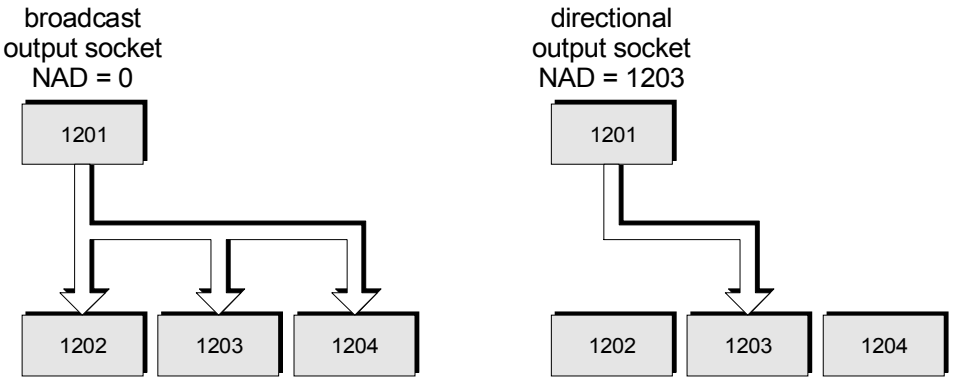
Sockets

Socket is the set of variables which VCP transmits or receives through the network. Every VCP can contain more input and more output sockets. Total number of sockets per device is limited to 32. Every socket can contain up to 256 variables. Variables from the output socket will be distributed to the network. Variables from the input socket will be received when addressed VCP transmits its output socket.



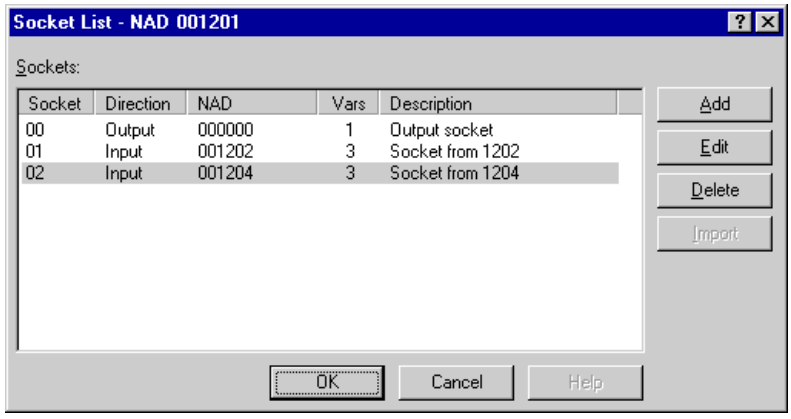
Socket consists of a list of variables and of network address. Matched input and output sockets should have the same structure, concerning type and order of variables. Names of variables in the input and the output socket may be different, but it is much better to keep them identical.

Output socket may have (and usually does have) the network address zero, which means that this socket is broadcasted to all devices that listen to it. If one output socket is broadcasted, no other output sockets on the same device are allowed to be broadcasted, because the sockets are identified by the sender address, and in that case receiver cannot distinguish between broadcast and directional socket.

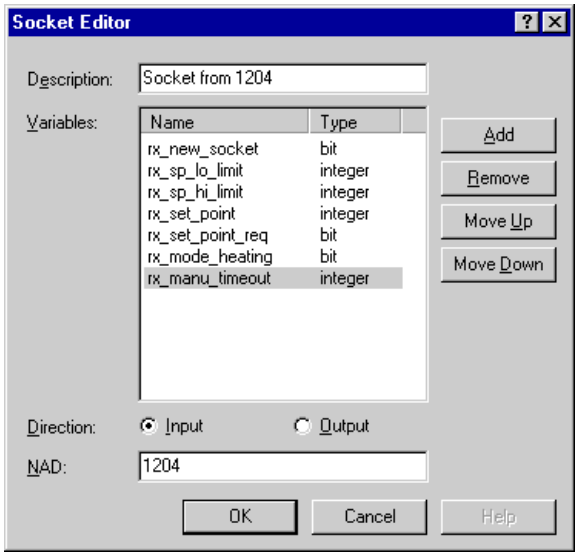


Variables declared in the input socket should be used as read-only. If a value is assigned in the program, a first received socket will change it. This may be used to detect an incoming socket.

To work with sockets open the Socket list dialog box by selecting View/Socket Editor (**F8**).



To add a new socket press the Add button. To edit selected socket, press the Edit button.

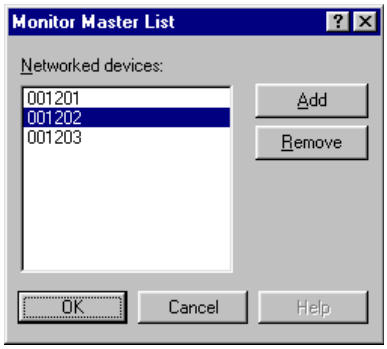


To enter a new variable, press the Add button. To rearrange variable order, drag-and-drop the desired variable, or use Move Up and Move Down buttons.

Debug mode

CyPro Variable Monitor may be used to debug the A-bus network. In that case, the variable monitor performs network mastering. The VCP master should be disabled while debugging, since only one master can run in the network.

To define Monitor Master List, press Master List button and enter NADs of connected devices.

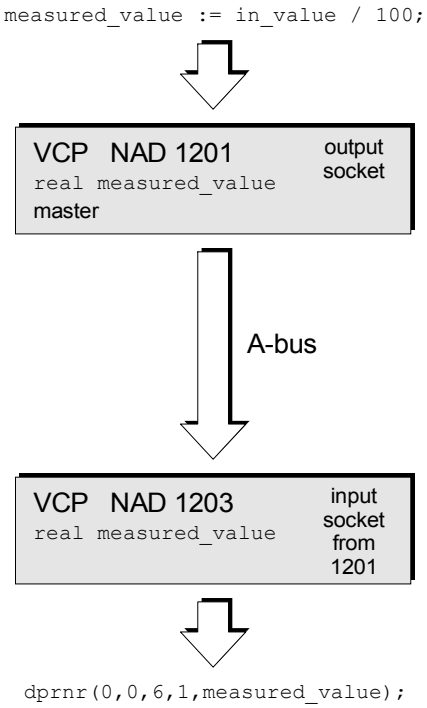


To start the network, check the Mastering checkbox. When started, Variable Monitor will concurrently display variables from the selected VCP and perform a network mastering.

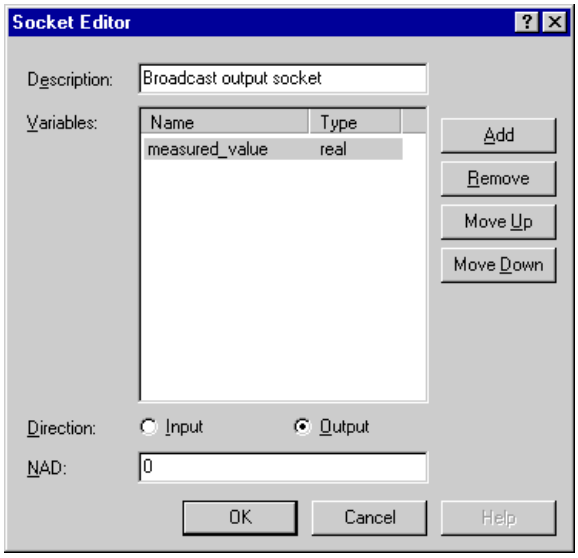
In a huge network, it is useful to run only a part of the network, while the other communication is stopped. In such case, put only the desired VCPs to the master list.

Network example

The following example illustrates a very simple A-bus network. The network consists of two VCPs, 1201 and 1203. Each VCP has one socket with one variable. Value assigned to the measured_value in 1201 is transported to the 1203. 1203 displays measured value on the operator panel.

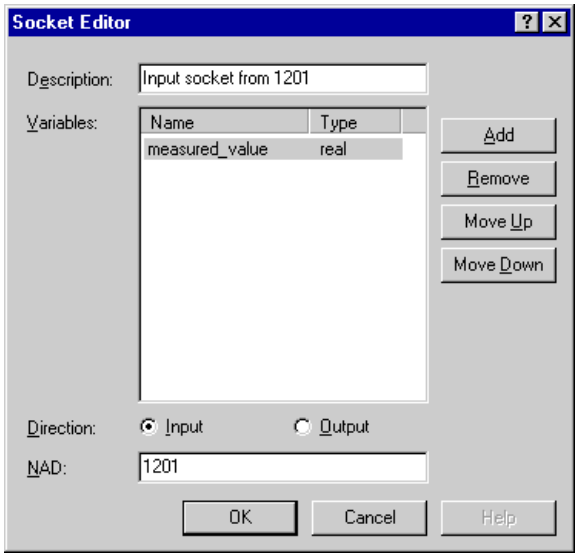


Here is how the 1201 output socket is created:

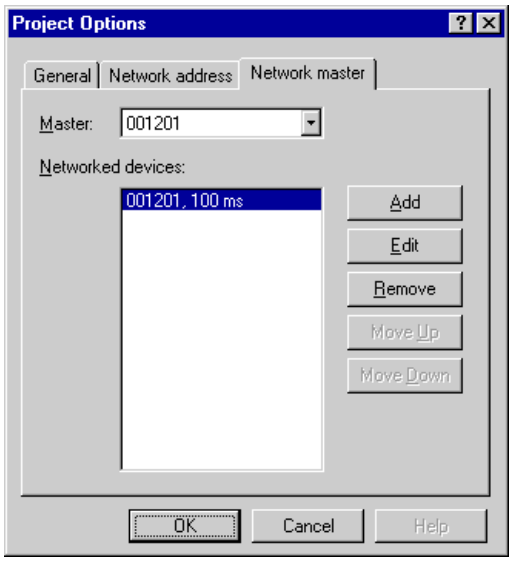


The NAD of the output socket is zero, so the socket is broadcasted. The other possibility is to set NAD to 1203, so the socket will be sent to 1203 only. If only two VCPs are connected, two variations will be similar, but if a number of VCPs communicate, the broadcast option should be used carefully.

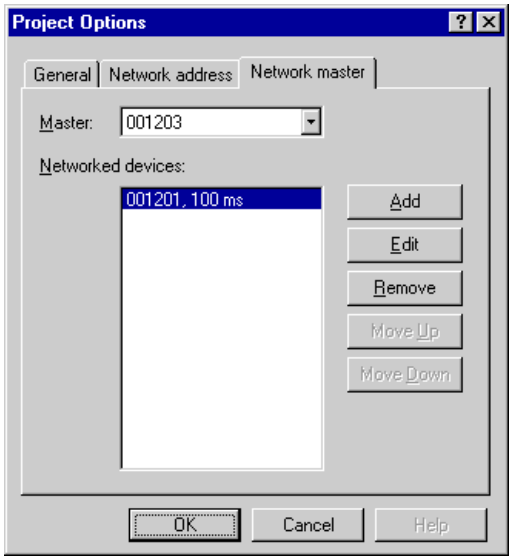
The input socket of 1203 has NAD 1201, to indicate that the data will be received from the 1201. NAD of the input socket should never be zero.



The network master is 1201. List of networked devices also contains 1201, so in this case it is actually the VCP that asks itself to transmit the data.



The master also may be 1203. In that case the program should have the network master set to 1203 and should contain 1201 in the network device list.

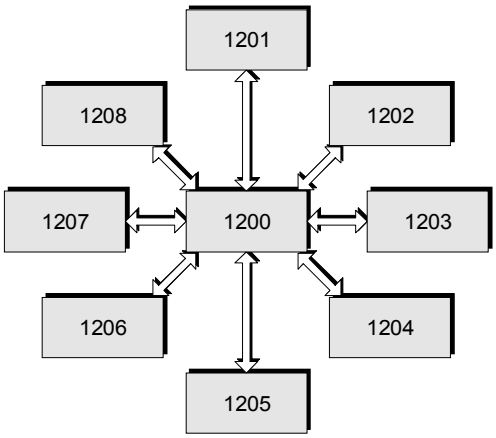


If the network master is 1203, the 1201 master should be disabled.

Network performance

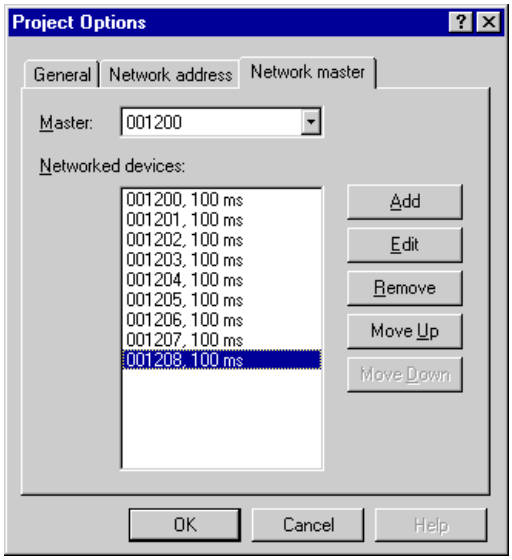
When network traffic becomes larger, an affort should be put to achieve optimal network performance.

Consider the following example:



Each VCP, except the central one, has one input and one output socket. The central VCP (1200) has eight input and eight output sockets. It collects all measurements and controls other VCPs.

Network master list may look like this:



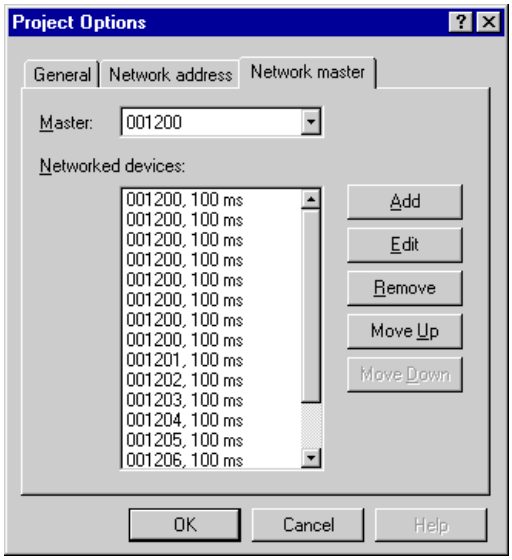
In that case, socket traffic will flow according to the following chart:

1. 1200 ⇨ 1201	10. 1200 ⇨ 1202	19. 1200 ⇨ 1203	28. 1200 ⇨ 1204
2. 1201 ⇨ 1200	11. 1201 ⇨ 1200	20. 1201 ⇨ 1200	29. 1201 ⇨ 1200
3. 1202 ⇨ 1200	12. 1202 ⇨ 1200	21. 1202 ⇨ 1200	30. 1202 ⇨ 1200
4. 1203 ⇨ 1200	13. 1203 ⇨ 1200	22. 1203 ⇨ 1200	31. 1203 ⇨ 1200
5. 1204 ⇨ 1200	14. 1204 ⇨ 1200	23. 1204 ⇨ 1200	32. 1204 ⇨ 1200
6. 1205 ⇨ 1200	15. 1205 ⇨ 1200	24. 1205 ⇨ 1200	…
7. 1206 ⇨ 1200	16. 1206 ⇨ 1200	25. 1206 ⇨ 1200	
8. 1207 ⇨ 1200	17. 1207 ⇨ 1200	26. 1207 ⇨ 1200	
9. 1208 ⇨ 1200	18. 1208 ⇨ 1200	27. 1208 ⇨ 1200	

All sockets will be transmitted and received in 9×8=72 communication cycles. Duration of one communication cycle depends of socket size and scan time of the corresponding VCPs (master, transmitter and receiver), but for a moderately loaded VCP 50ms would be a fairly good estimate. In that case, complete network update time will be 72×50ms=3.6s, which can hardly be considered as “warp speed”.

To achieve optimal network response time, network master list should contain multiple references to the VCP that has more output sockets. Number of references should be equal to the number of output sockets.

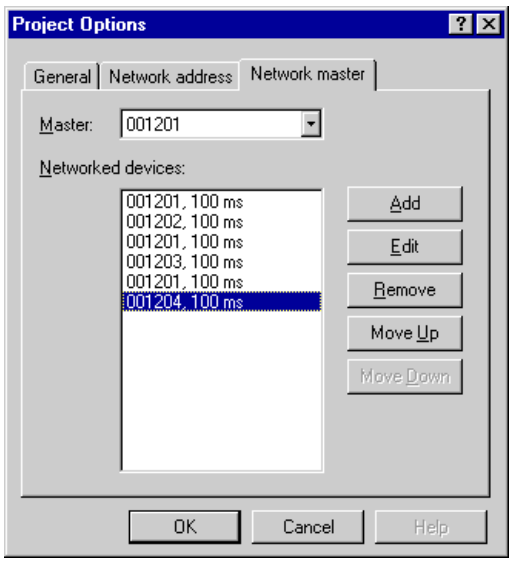
In our example, master should have the address 1200 repeated eight times, one for each output socket. Other VCPs are listed only once, since they have only one output socket.



Socket traffic is more optimized now. All sockets will be transmitted and received in 8+8=16 communication cycles. Complete network response time falls to 16×50ms=800ms, which is 4.5 times faster.

Even better response time may be achieved by converting all directional output sockets into one large broadcast socket. The disadvantage of this solution is that the other VCPs have to select their control data from the broadcast socket, so each VCP should have its own program.

Multiple references may be used also when the priority of one particular VCP in the network is increased. The following example shows the network of four VCPs, with the increased priority of 1201 output sockets.



Network throughput may also be affected by choosing the network master. For best results the network master should execute a short PLC program, resulting in a high scan frequency. To check the scan frequency, enable the internal variable scan_frequency.

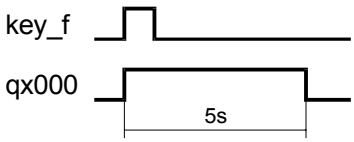
Also some improvement may be made if the network master is VCP with a maximum number of output sockets, because the master handles its own output sockets slightly faster than the output sockets of the other VCPs.

VCP tutorial

Your first VCP program

Step one: define problem

In the first example, we will take a very simple task: a timer activated by a key press. By pressing a key, timer will turn the output on for a predefined time period, about 5 seconds.

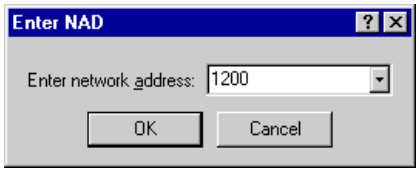


A key for activating the timer can be any binary input, but it is simpler to use one of the display keys. The output is the first binary output, qx000.

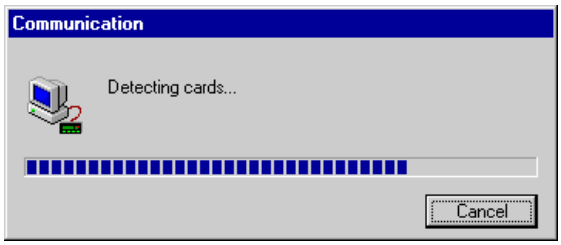
Step two: select hardware

Connect CyBro and PC with a serial cable.

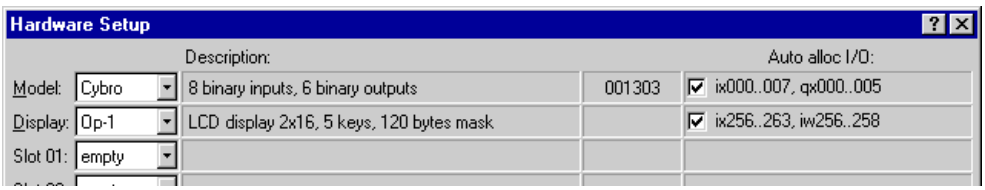
Start CyPro and select New Project. Open Hardware Setup and start Autodetect. A small dialog box will appear, asking to enter VCP address.



Enter serial number of connected VCP and press OK. Serial number will appear in the status bar.



Autodetect procedure will identify all properly connected IEX modules. In our example, it will detect CyBro and the operator panel.



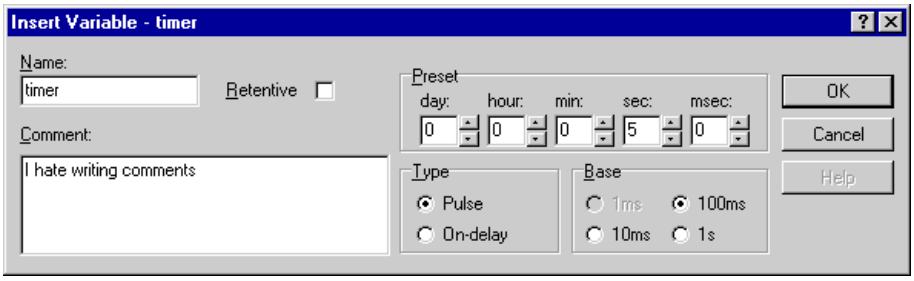
Leave the Auto alloc checkbox marked, so that all input and output variables could be allocated automatically.

To finish, click the OK button.

Step three: allocate variables

Our simple project needs one timer type variable.

Start Allocation Editor, select timer tab and press Insert. Insert Variable dialog box will appear:



Enter name, adjust preset value, choose pulse type, select 100ms base and press OK.

Step four: write code

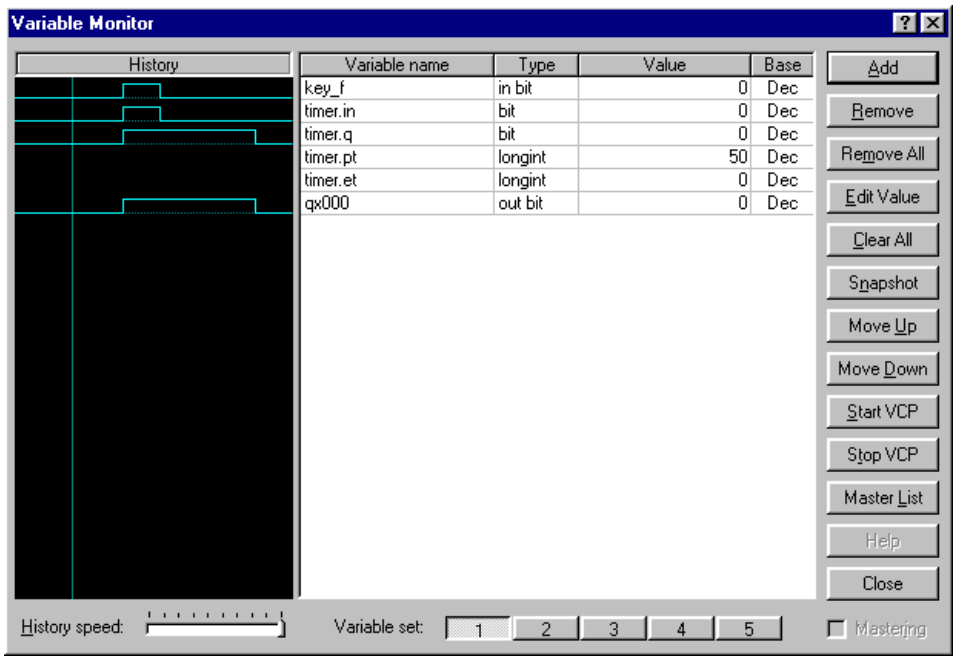
PLC code should connect the timer input to the key and the timer output to the output relay. This can be done by:

```
timer.in:=key_f;
qx000:=timer.q;
```

Step five: bring it to life

To compile and transfer the program to the VCP, press Start button. Status line indicator will show that the program is running.

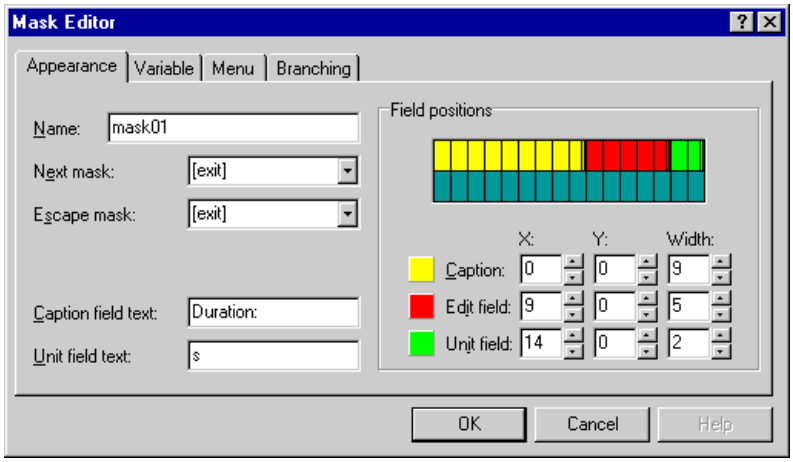
To examine the whole action, start Variable Monitor and Add allocated variables.



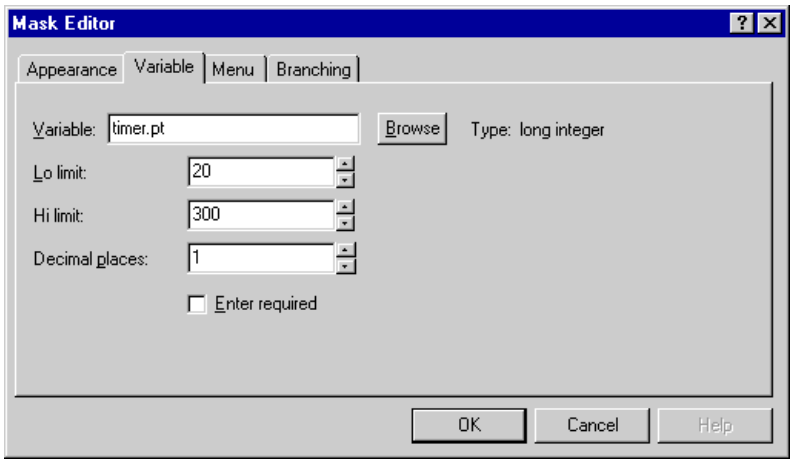
Step six: new frontiers

Another challenge would be to make the timer adjustable.

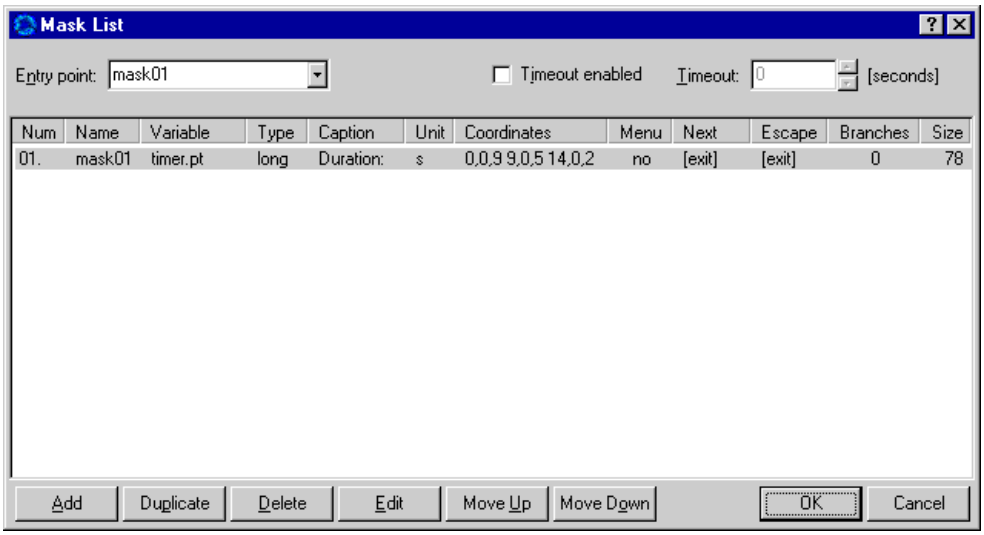
A way to accomplish this is the mask system. Start Mask Editor and press Add to create a new mask. Enter caption and unit strings. Other fields may remain default.



Switch to Variable tab, enter timer.pt variable and adjust boundaries. Note that timer resolution is 100ms, so one decimal place is required to properly indicate seconds.

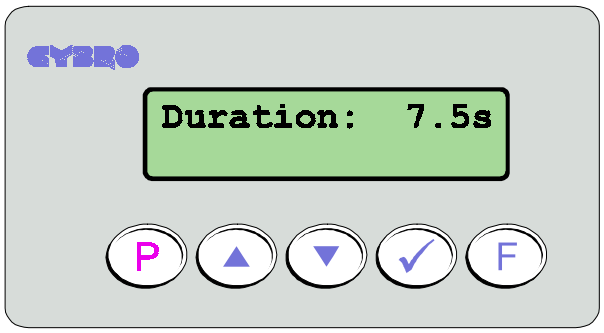


Press OK. Set entry point to mask01 and press OK again.



Press Start button. The program is automatically compiled, transferred and started.

To adjust timer value, press the **P** key, adjust with **up** and **dn** and exit with **P** again.



To activate timer press the **F** key.

Appendix

Data types summary

Elementary

type	width	range
bit	1	0..1
word	16	-
integer	16	-32768..32767
long	32	-2147483648..2147483647
real	32	$-3.4x10^{38}..3.4x10^{38}$

Input/Output

type	width	equal type	description
in bit	1	bit	binary input
out bit	1	bit	binary output
in word	16	integer	analog input
out word	16	integer	analog output

Timer

field	type	direction	description
in	bit	input	input
pt	long	input	preset time
et	long	output	elapsed time
q	bit	output	output

Counter

field	type	direction	description
cu	bit	input	count up input
cd	bit	input	count down input
ld	bit	input	load input
r	bit	input	reset input
pv	integer	input	preset value
cv	integer	input	counter value
qu	bit	output	up limit output
qd	bit	output	down limit output

Internal variables

name	type	direction	description
first_scan	bit	read only	active during first scan only
scan_overrun	bit	read only	scan timeout occurred
clock_10ms	bit	read only	10ms clock
clock_100ms	bit	read only	100ms clock
clock_1s	bit	read only	1s clock
clock_1min	bit	read only	1min clock
all_outputs_off	bit	read/write	if active, all outputs go off
retentive_fail	bit	read only	indicates that retentive memory has failed
scan_time	int	read only	last scan execution time [ms]
scan_time_max	int	read only	max. scan execution time encountered [ms]
scan_frequency	int	read only	number of scans per second
io_refresh_time	int	read only	last i/o refresh time [ms]
io_refresh_time_max	int	read only	maximal i/o refresh time encountered [ms]

Instruction list summary

Instructions

Move

ld	move variable or constant to accumulator
ldn	move complement of variable to accumulator
st	move accumulator to variable
stn	move complement of accumulator to variable
set	set accumulator or variable
setc	if condition true set variable
res	clear accumulator or variable
resc	if condition true clear variable

Logic

cpl	complement accumulator or variable
and	logical and accumulator with variable or constant
andn	logical and accumulator with complement of variable or constant
or	logical or accumulator with variable or constant
orn	logical or accumulator with complement of variable or constant
xor	exclusive or accumulator with variable or constant
xorn	exclusive or accumulator with complement of variable or constant
shl	shift left accumulator, set LSB to zero
shr	shift right accumulator, set MSB to zero
rol	rotate left accumulator, copy MSB to LSB
ror	rotate right accumulator, copy LSB to MSB
fp	positive flank, 1 if low-to-high transition detected, 0 otherwise
fn	negative flank, 1 if high-to-low transition detected, 0 otherwise

Arithmetic

neg	change sign of accumulator
add	add variable or constant to accumulator
sub	subtract variable or constant from accumulator
mul	multiply accumulator with variable or constant
div	divide accumulator with variable or constant
mod	remains of dividing accumulator with variable or constant

Compare

eq	test if accumulator equal to value
ne	test if accumulator not equal to value
gt	test if accumulator greater then value
ge	test if accumulator greater or equal value
lt	test if accumulator lower then value
le	test if accumulator lower or equal value

Branch

jmp label	unconditional jump to position indicated by label
jmpc label	jump if condition true
jmpnc label	jump if condition not true
cal subroutine	call subroutine
calc subroutine	call subroutine if condition is true
calnc subroutine	call subroutine if condition is not true

Type conversions:

xtoy convert acc from type x to type y (bit, word, integer, long, real)

Allowed type conversions

	bit	word	int	long	real
bit		+	+	+	
word			+	+	
int		+		+	+
long		+	+		+
real			+	+	

Allowed type combinations

	bit	word	int	long	real	acc	const	var
ld	+	+	+	+	+		+	+
ldn	+							+
st	+	+	+	+	+			+
stn	+							+
set	+					+		+
setc	+							+
res	+					+		+
resc	+							+
cpl	+	+				+		+
and	+	+					+	+
andn	+	+					+	+
or	+	+					+	+
orn	+	+					+	+
xor	+	+					+	+
xorn	+	+					+	+
shl		+				+		
shr		+				+		
rol		+				+		
ror		+				+		
fp	+					+		+
fn	+					+		+
neg			+	+	+	+		
add			+	+	+		+	+
sub			+	+	+		+	+
mul			+	+	+		+	+
div			+	+	+		+	+
mod			+	+			+	+
eq	+	+	+	+	+		+	+
ne	+	+	+	+	+		+	+
gt			+	+	+		+	+
ge			+	+	+		+	+
lt			+	+	+		+	+
le			+	+	+		+	+
jmp							+	
jmpc							+	
jmpnc							+	
cal							+	
calc							+	
calnc							+	
x-to-y	+	+	+	+	+	+		
dprnx	+		+	+	+			+

Structured text summary

Operators

operator	alias	unary	binary	bit	word	int	long	real	result
+			•			•	•	•	same
-		•	•			•	•	•	same
*			•			•	•	•	same
/			•			•	•	•	same
mod	%		•			•	•		same
not	!	•		•	•				same
and	&		•	•	•				same
or	|		•	•	•				same
xor			•	•	•				same
=	==		•	•	•	•	•	•	bit
<>	!=		•	•	•	•	•	•	bit
<			•			•	•	•	bit
<=			•			•	•	•	bit
>			•			•	•	•	bit
>=			•			•	•	•	bit
:=			•	•	•	•	•	•	same

Statements

if...then...else

```
if <expression> then
  <statements>;
elsif <expression> then
  <statements>;
else
  <statements>;
end_if;
```

case...of

```
case <expression> of
  <value1>: <statements>;
  <value2>: <statements>;
  ...
  <valuen>: <statements>;
else
  <statements>;
end_case;
```

for...do

```
for <var>:=<expression> to <expression> do
  <statements>;
end_for;
```

while...do

```
while <expression> do
  <statements>;
end_while;
```

Edge detect functions

positive edge detect

```
fp(b:bit):bit;
```

negative edge detect

```
fn(b:bit):bit;
```

Cast functions

```
int(expression):int;
word(expression):word;
long(expression):long;
real(expression):real;
```

Display functions

clear display

```
dclr();
```

print ASCII character

```
dprnc(x:int, y:int, c:char);
```

print string

```
dprns(x:int, y:int, str:string);
```

print binary value

```
dprnb(x:int, y:int, c0:char, c1:char, value:bit);
```

print integer value

```
dprni(x:int, y:int, width:int, zeroblank:bit, value:int);
```

print long value

```
dprnl(x:int, y:int, width:int, zeroblank:bit, value:long);
```

print real value

```
dprnr(x:int, y:int, width:int, dec:int, value:real);
```

Legend:

x.....x position (0-left)
y.....y position (0-top)
width .....print width
zeroblank....zero blanking (0-no, 1-yes)
dec.....decimal places
c.....single character
str.....array of characters
value .....value to print

Com2 functions

port initialization

```
com_init(baud_rate:long, data_bits:int, parity:int, stop_bits:int):bit;
```

transmit

```
tx_start(char_num:int);
tx_stop();
tx_count():int;
tx_active():bit;
```

receive

```
rx_start(beg_ch:char, end_ch:char, len:int, msg_tout:int, char_tout:int);
rx_stop();
rx_count():int;
rx_active():bit;
rx_status():int;
```

parse received message

```
rx_bufrd(position:int):int;
rx_strcmp(position:int, str:string):bit;
rx_strpos(position:int, str:string):int;
rx_strtoi(position:int):int;
rx_strtol(position:int):long;
rx_strtor(position:int):real;
```

High speed counter functions

start/stop counting

```
hsc_start();
hsc_stop();
hsc_active():bit;
```

read/write counter value

```
hsc_read():long;
hsc_write(position:long);
```

set/reset high speed action

```
hsc_set_action(action:bit, variable:bit);
hsc_reset_action();
hsc_check_action():bit;
```

enable/disable counter reset on zero input

```
hsc_enable_zero();
hsc_disable_zero();
hsc_check_zero():bit;
```

detect and read position of zero input

```
hsc_detect_zero():bit;
hsc_read_zero():long;
```

Real-time clock functions

read date/time

```
rtc_read_year():int;
rtc_read_month():int;
rtc_read_date():int;
rtc_read_weekday():int;
rtc_read_hour():int;
rtc_read_min():int;
rtc_read_sec():int;
```

write date/time

```
rtc_write_year(year:int);
rtc_write_month(month:int);
rtc_write_date(date:int);
rtc_write_weekday(weekday:int);
rtc_write_hour(hour:int);
rtc_write_min(min:int);
rtc_write_sec(sec:int);
```

Error messages

PLC status

1	Internal error.
2	Invalid PLC program.
3	Scan overrun (congestion error).

Compiler messages

1	Invalid instruction.
2	Invalid variable type.
3	Undefined symbol.
4	Extra parameter.
5	Illegal binary constant.
6	Illegal hexadecimal constant.
7	Illegal constant.
8	Constant out of range.
9	Illegal number base.
10	Duplicate label.
11	Bit type constant expected.
12	Word type constant expected.
13	Integer type constant expected.
14	Long integer type constant expected.
15	Real type constant expected.
16	Insufficient parameters.
17	Invalid label.
18	Illegal operation with label.
19	Illegal operation with subroutine.
20	Bit type variable expected.
21	Word type variable expected.
22	Integer type variable expected.
23	Long integer type variable expected.
24	Real type variable expected.
25	String constant expected.
26	Closing quotes expected.
27	Invalid quote placement.
28	Label buffer overrun.
29	Jump buffer overrun.
30	Call buffer overrun.
31	PseudoCode buffer overrun.
32	Display not defined in hardware setup.
33	Invalid char.
34	Invalid number of decimal places.
35	Illegal operation with constant.
36	Illegal operation with variable.
37	Illegal operation with argument.
38	Cannot define accumulator type.
39	Invalid value type in accumulator.
40	Value must be loaded in accumulator.
41	Instruction cannot operate with this accumulator type.
42	Internal error: cannot handle argument.
43	Printing out of display boundaries.
43	‘)’ expected.
45	Cannot promote type.
46	Variable expected.
47	Syntax error.
48	'THEN' expected.
49	Boolean expression expected.
50	No instructions allowed after 'THEN'.

51	'END_IF' expected.
52	Mismatched 'END_IF'.
53	No instructions allowed after 'ELSE'.
54	Invalid label placement.
55	'OF' expected.
56	Boolean, integer or long integer expression expected.
57	No instructions allowed after 'OF'.
58	Duplicate 'ELSE' statement.
59	'END_CASE' expected
60	Mismatched 'END_CASE'.
61	Empty 'CASE' construction.
62	'(' expected
63	Long integer expression expected
64	'[' expected
65	']' expected
66	Integer expression expected
67	'DO' expected
68	No instructions allowed after 'DO'
69	'END_WHILE' expected
70	Mismatched 'END_WHILE'
71	'TO' expected
72	'END_FOR' expected
73	Mismatched 'END_FOR'
74	Integer or long integer type variable expected
75	Assign operator expected
76	Missing semicolon
77	':' expected
78	Real expression expected
79	Function doesn't returns result
80	Constant expected
81	Statement expected
82	Function parameter expected

Allocation messages

100	Variable allocated out of I/O range.

Mask messages

200	Display not defined in hardware setup.
201	Undefined variable ‘%s’.
202	Caption ‘%s’ too long. Predefined length is %d.
203	Input value doesn’t fit into edit field.
204	Menu item ‘%s’ doesn’t fit into edit field.
205	Min. value greater then max. value.
206	Caption and edit field intersection.
207	Mask memory overload.
208	Unknown enter location: %s.
209	Unknown escape location: %s.
210	Unknown branching location: %s.
211	Unknown entry point: %s.

Socket messages

300	Variable ‘%s’ doesn’t exists.
301	Undefined socked receive NAD.
302	Socket has no defined variables.

Keyboard shortcuts

Common

F1		Help
Shift+F1		What's this
F2		Syntax check
F5		Hardware setup
F6		Allocation editor
F7		Mask editor
F8		Socket editor
F9		Send project to VCP
F10		On-line variable monitor
F11		Start VCP
F12		Stop VCP
Ctrl+O		Open project
Ctrl+S		Save project
Ctrl+P		Print project
Ctrl+D		Communication address popup menu
Ctrl+M		Set next communication address
Ctrl+N		Set previous communication address
Ctrl+1		Move focus to project tree
Ctrl+2		Move focus to edit window
Ctrl+3		Move focus to message list
Ins		Context sensitive insert
Delete		Context sensitive delete
Ctrl+Up		Move list item up
Ctrl+Dn		Move list item down
Ctrl+F6	Ctrl+Tab	Next window
Ctrl+Shift+F6	Ctrl+Shift+Tab	Previous window
Ctrl+F4		Close window
Alt+F4		Exit program

Text editor

Ctrl+Z	Alt+Backspace	Undo
Shift+Ctrl+Z		Redo
Ctrl+X	Shift+Del	Cut
Ctrl+C	Ctrl+Insert	Copy
Ctrl+V	Shift+Insert	Paste
Ctrl+A		Select all
Ctrl+F		Find
F3		Find next
Ctrl+R		Find & replace
Ctrl+W	Alt+Insert	Insert/overwrite
Ctrl+G		Go to line
Ctrl+B		Match brackets

Ctrl+I	Indent block
Ctrl+U	Unindent block
Ctrl+Alt+C	Comment selected block
Ctrl+Alt+U	Uncomment selected block